Cuneiform
A Functional Language for Large-Scale Data Analysis

D i s s e r t a t i o n

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von

Jörgen Brandt, M.Sc.

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschafltichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter/innen:

1. Ulf Leser
2. Wolfgang Reisig
3. Adelinde Uhrmacher

Tag der mündlichen Prüfung:
04.12.2020

Abstract

Bioinformatics and next-generation sequencing data analyses often form large and complex pipelines. The tools and libraries making up the processing steps in these pipelines come from different sources and have different interfaces which hampers integrating them into data analysis frameworks. Also, these pipelines process large data sets. Thus, users need to parallelize independent processing steps. The state of the art in large-scale scientific data analysis for bioinformatics and next-generation sequencing are scientific workflow systems. A scientific workflow system allows researchers to describe a data analysis pipeline as a scientific workflow which integrates external software, defines the data dependencies forming a data analysis pipeline, and parallelizes independent processing steps. Scientific workflow systems consist of a workflow language providing a user interface, and an execution environment. The workflow language determines how users express workflows, reuse and compose workflow fragments, integrate external software, how the scientific workflow system identifies independent processing steps, and how we derive optimizations from a workflow's structure. The execution environment schedules and runs data processing operations.

In this thesis we present Cuneiform, a workflow language, and its distributed execution environment. For Cuneiform's design we take the perspective of programming languages. We adopt methods from functional programming towards composition and expressing data dependencies. We apply operational semantics and type systems to define well-formedness, consistency, and reduction of Cuneiform workflows. For the design of the distributed execution environment we take the perspective of distributed systems. We apply Petri nets to define the communication patterns among the distributed execution environment's agents.

We show how to use Cuneiform to (i) integrate foreign tools and libraries from languages like R or Python by wrapping them in functions, (ii) create complex workflows by composing simple workflows, and (iii) use language features common in functional programming like conditional execution, iteration, folding, recursion, or higher-order functions in the context of bioinformatics and next-generation sequencing applications. The execution environment underlying Cuneiform distributes independent foreign function applications to run these applications in parallel.

Some contemporary workflow languages like Swift or Nextflow also promote a functional style. However, to our knowledge, Cuneiform is the only external, statically typed workflow language providing bioinformaticians advanced features of functional programming in a distributed setting.

Zusammenfassung

In der Bioinformatik und der Next-Generation Sequenzierung benötigen wir oft große und komplexe Verarbeitungsabläufe um Daten zu analysieren. Die Werkzeuge und Bibliotheken, die hierin die Verarbeitungsschritte bilden, stammen aus unterschiedlichen Quellen und exponieren unterschiedliche Schnittstellen, was ihre Integration in Datenanalyseplattformen erschwert. Hinzu kommt, dass diese Verarbeitungsabläufe meist große Datenmengen prozessieren weshalb Forscher erwarten, dass unabhängige Verarbeitungsschritte parallel laufen. Der Stand der Technik im Feld der wissenschaftlichen Datenverarbeitung für Bioinformatik und Next-Generation Sequenzierung sind wissenschaftliche Workflowsysteme. Ein wissenschaftliches Workflowsystem erlaubt es Forschern Verarbeitungsabläufe als Workflow auszudrücken. Solch ein Workflow erfasst die Datenabhängigkeiten in einem Verarbeitungsablauf, integriert externe Software und erlaubt es unabhängige Verarbeitungsschritte zu erkennen, um sie parallel auszuführen.

In dieser Arbeit präsentieren wir Cuneiform, eine Workflowsprache, und ihre verteilte Ausführungsumgebung. Für Cuneiform's Design nehmen wir die Perspektive der Programmiersprachentheorie ein. Wir lassen Methoden der funktionalen Programmierung einfließen um Komposition und Datenabhängigkeiten auszudrücken. Wir nutzen operationelle Semantiken um zu definieren, wann ein Workflow wohlgeformt und konsistent ist und um Reduktion zu erklären. Für das Design der verteilten Ausführungsumgebung nehmen wir die Perspektive der verteilten Systeme ein. Wir nutzen Petri Netze um die Kommunikationsstruktur der im System beteiligten Agenten zu erklären.

Wir zeigen wie wir Cuneiform nutzen um (i) externe Bibliotheken aus R oder Python zu integrieren und als Funktionen auszudrücken, (ii) komplexe Workflows aus einfachen Workflows zu komponieren und (iii) funktionale Programmiertechniken anzuwenden, beispielsweise bedingte Ausführung, Iteration, Faltung, Rekursion, oder Funktionen höherer Ordnung. Dies zeigen wir in Anwendungsfällen aus der Bioinformatik und der Next-Generation Sequenzierung.

Während gängige Workflowsprachen wie Swift oder Nextflow ebenfalls einen funktionalen Stil andeuten ist, unseres Wissens nach, Cuneiform die einzige externe, statisch getypte Workflowsprache die es Bioinformatikern erlaubt fortgeschrittene funktionale Programmierung in einer verteilten Ausführungsumbebung anzubieten.

Contents

List of Figures

1. Introduction

Over the past years, the the cost of data acquisition in bioinformatics and genomics has been declining [156, 181]. Scientists have been tapping high-throughput methods to generate data [175, 195] and, thus, also need high throughput methods to analyze this data [200]. This decline in cost and increase in available data has enabled large studies like the Thousand Genomes Project [45]. Such large scale studies are feasible because researchers parallelize data processing and distribute work to many computers [18, 107]. As studies grow in scale, also the tool chains for analyzing data have grown longer and more complex. Today we need to analyze larger data sets, apply more analysis steps, integrate more software tools, and deal with more complex data flows than a decade ago. In addition, bioinformatics tools expose a variety of programming interfaces in different programming languages which complicates their integration. For instance, a read mapper may provide a command-line interface while a visualization library may provide a programming interface in R. Thus, researchers need data analysis tools enabling them to express complex data analyses, to integrate external tools and libraries, and to use parallel and distributed compute resources to keep up with the rate at which researchers generate new data and conceive new software.

The state of the art platforms for bioinformatics and genomics data analysis are scientific workflows systems [212]. A scientific workflow integrates external software to define processing steps and specifies the data dependencies among these steps [245]. Bundled with external software and data, a scientific workflow is an exhaustive description of a data analysis procedure that researchers can share and reproduce [49]. Running a workflow reveals concrete processing steps which form a directed acyclic graph. Such a graph documents which processing steps have produced which data, i.e., it documents the provenance of all artifacts (data and processing steps) that take part in the workflow [48, 207]. One distinguishing feature of scientific workflows is their focus on integrating external software. Some workflow systems integrate only command-line tools, e.g., Pegasus DAX. Other workflow languages, like Nextflow, integrate libraries with programming interfaces in many different programming languages. In addition, scientific workflow languages often offer abstraction features. Some workflow languages, like Pegasus DAX, use abstraction only sparingly, letting the user spell out the directed acyclic graph that defines data dependencies. This has the advantage that static scheduling schemes can be applied [119]. Other workflow languages, like Swift, offer high-level language features to describe a workflow abstractly. This has the advantage that users can compose complex workflows that depend on knowledge available only at runtime and nest workflows in sub-workflows to describe complex workflows and to reuse common processing patterns.

Another important aspect is parallelization. The data flow orientation of scientific

workflows allows independent processing steps to run in parallel and to distribute work to many computers. Some workflow systems like BioPig [171] build upon large scale data processing platforms specifically to distribute data processing to many computers. Other workflow systems like KNIME [21] were designed to run on only one machine but later extended to run in compute clusters [118].

To design and build a scientific workflow language we need to create models that convey our understanding about that language. One possible model for scientific workflows is the lambda calculus [127, 128]. The lambda calculus allows us to express the data dependencies among processing steps as function applications and define the meaning of scientific workflows using operational semantics. We can program scientific workflows much like we program functional programs and we can design a scientific workflow language much like we design a functional programming language. Furthermore, we can use type theory to statically analyse scientific workflows as shown, e.g., by Taverna [225]. However, defining the meaning of a scientific workflow language is not enough to understand scientific workflow systems as distributed systems. Many aspects of a distributed workflow system are independent of the language and its meaning: e.g., how a scientific workflow system schedules processing steps to its workers, how it interacts with a distributed file system, how it caches intermediate results, or how it maintains a consistent state facing partial failure. Thus, we need to define the communication patterns that govern a scientific workflow system in addition to the meaning of the workflow language.

In this thesis we present Cuneiform, a functional scientific workflow language and its distributed execution environment. Like a functional program, a Cuneiform program is an expression composed of functions, function applications, variables, and literal data. In the absence of mutable state a Cuneiform expression's subexpressions are independent which enables parallel execution. In Cuneiform, a workflow task template is described as a function, a concrete workflow task is described as a function application, and a filename is described as literal data. From these building blocks users compose complex workflows. In addition, Cuneiform allows to define foreign functions to integrate external software. This way Cuneiform allows users to create complex workflows that integrate external software as functional programs. Cuneiform's distributed runtime environment runs independent workflow tasks in parallel on many computers.

We define Cuneiform's abstract syntax, give a simple type system, and define its semantics using reduction semantics. To run such a functional workflow description in a distributed environment we give Cuneiform's language semantics in a way that accomodates communication with a distributed execution environment. We define the distributed execution environment in terms of its communication patterns using Petri nets. In sum, we provide two complementary models: one model specifying Cuneiform and its semantics and one specifying its distributed execution environment and its communication patterns. As a result, Cuneiform provides the same readability, expressiveness, safety, composability, and capability to abstract as a functional programming language. Next we describe the implementation of the Cuneiform interpreter and its distributed execution environment.

1.1. Contribution

In this thesis we present Cuneiform, a distributable, functional language for scientific workflows. We contribute Cuneiform and a compatible distributed execution environment in both specification and implementation. Furthermore, we discuss several applications from bioinformatics and next-generation sequencing.

To discuss Cuneiform, we give a language specification in the form of an abstract syntax. Using this abstract syntax we give Cuneiform's type system and provide a reduction semantics that formally specifies the meaning of a Cuneiform script and how a Cuneiform interpreter communicates with a compatible distributed execution environment.

Also, we give a specification of a simple but complete distributed execution environment in the form of a Petri net model. We specify communication patterns, scheduling, caching, and failure recovery.

We implement both the Cuneiform interpreter and the compatible distributed execution environment. The interpreter implementation comprises a scanner, a parser, a type system, and a runtime system for Cuneiform. The implementation of the distributed execution environment comprises two types of services: (i) a scheduler component and (ii) a worker component. Herein, the scheduler is a central service coordinating its attached workers. We use the Erlang programming language for the implementation of these services.

Finally, we study the aptitude of Cuneiform in five different applications from bioinformatics and next-generation sequencing. In these applications we show that Cuneiform is capable to express complex real-life workflows, that it parallelizes independent tasks in a distributed compute environment, and that it integrates software with different programming interfaces. We analyze execution logs in the context of one of these applications to understand how the distributed execution environment use resources like time, CPU power, network bandwidth, and input/output in a medium-sized cluster.

Overall, we contribute the design and implementation of a scalable scientific workflow system based on functional programming.

1.2. Thesis Outline

We structure the remainder of this thesis as follows: Chapter 2 gives an overview over the research areas we touch. We introduce genomics as a field that relies on large-scale data analysis. We discuss genomic disciplines like variant detection or ChIP-Seq and explain why these disciplines are relevant and what distinguishes them regarding their data analysis needs. We also discuss the challenges that appear analyzing large data sets. These challenges motivate Cuneiform's design. We introduce scientific workflows as the state of the art in analyzing large-scale data sets in bioinformatics. Finally, we give an overview over operational semantics, a method for specifying programming languages, and Petri nets, a method for specifying distributed systems.

Chapter 3 explains Cuneiform as a programming language. First, we give a quick tour discussing important Cuneiform features by example. We demonstrate how Cu-

neiform achieves core workflow aspects such as automation and reproducibility. Also, we demonstrate how external software can be integrated in Cuneiform. Furthermore, we discuss how Cuneiform derives parallelism from Cuneiform scripts. Lastly, we discuss the consequences of Cuneiform being a functional programming language and detail how higher-order constructs like mapping and folding can be combined with recursion to specify classic algorithms such as quicksort or the Ackermann function. In the remainder of the section we define Cuneiform's semantics. We do so by first giving an abstract syntax and defining a type system to verify a workflow's consistency. Next, we define a reduction relation using reduction semantics which is a style of operational semantics. We demonstrate this reduction relation by applying it to small examples. Last, we discuss how an interpreter enacting Cuneiform's semantics interacts with the execution environment.

Chapter 4 explains Cuneiform as a distributed system. A distributed system defines the flow of information in a system of independent services. We describe Cuneiform's components and how they communicate both externally, exchanging messages with other components, and internally, arranging subcomponents to pursue a common goal. Cuneiform comprises three major components: (i) a client, interpreting a program matching Cuneiform's semantics, (ii) a scheduler, relaying function applications and their results between workers and clients, and (iii) a worker, performing a single, self-contained function application at a time reporting the result back to the scheduler.

Chapter 5 describes the software architecture of Cuneiform's implementation. We use distributed Erlang to build a scalable execution environment. We introduce a library to construct Erlang processes from a Petri net specification. In addition we describe how the Cuneiform language semantics are implemented in Erlang and how we generate a parser from Cuneiform's concrete syntax specification.

Chapter 6 describes several applications from bioinformatics or next-generation sequencing in which we used Cuneiform. We demonstrate a variant calling workflow and a ChIP-Seq workflow. In addition, we analyze execution logs from distributed runs and identify factors that contribute to fast workflow executions.

Chapter 7 provides a discussion and Chapter 8 concludes the thesis.

1.3. Own Prior Work

We have published parts of this thesis previously. Brandt et al. 2015 [31] introduce the Cuneiform programming language and its application areas. Jörgen Brandt wrote the sections enumerating Cuneiform's features, giving examples, and discussing a next-generation sequencing workflow in Cuneiform. Marc Bux wrote the sections introducing a Cuneiform-compatible distributed execution environment: Hi-WAY and discussing a performance experiment. Ulf Leser structured the text and contributed to introduction and conclusion sections.

Bux et al. 2015 [37] present Hi-WAY and Cuneiform in a demo covering a distributed k-means clustering use case. Marc Bux wrote the sections introducing Hi-WAY's and Cuneiform's joint architecture, covering Hi-WAY, and comparing the performance of a

variant calling workflow in Hi-WAY to an implementation in Tez. Jörgen Brandt wrote the sections introducing Cuneiform and discussing the k-means workflow. Ulf Leser structured the text and reviewed introduction and conclusion sections.

Bessani et al. 2015 [22] summarize the effort of the BiobankCloud project. Jörgen Brandt and Marc Bux wrote the section covering Hi-WAY and Cuneiform. Jim Dowling and Ulf Leser structured the text and wrote the introduction and conclusion sections.

Bux et al. 2017 [38] introduce the architecture of Hi-WAY, a language-agnostic workflow execution environment. Marc Bux wrote the sections describing how Hi-WAY schedules workflow tasks, how workflow languages integrate with Hi-WAY, and how work is distributed using Hadoop YARN. He also wrote the section discussing Hi-WAY's scalability in several performance experiments. Jörgen Brandt reviewed the workflow language interface section. Ulf Leser and Jim Dowling structured the text and reviewed introduction and conclusion sections.

Brandt et al. 2017 [33] present the Cuneiform language semantics as a structured operational semantics. Jörgen Brandt wrote the sections giving an example Cuneiform program from next-generation sequencing, defining the abstract syntax, defining a type system, and defining the reduction relation. Wolfgang Reisig reviewed the definitions. Ulf Leser reviewed introduction and example sections.

Brandt and Reisig 2018 [32] present an Erlang library to derive an Erlang process from a Petri net specification. Jörgen Brandt wrote the sections introducing Petri nets, describing the application programming interface, and discussing two examples: a worker pool manager and Cuneiform. Wolfgang Reisig reviewed the Petri net models.

Also, the Cuneiform website[1] and GitHub[2] make available documentation, examples, and design documents for Cuneiform that also appear in this thesis.

[1] `https://cuneiform-lang.org`
[2] `https://github.com/joergen7`

2. Background

Cuneiform's design touches several research domains: first, it touches its intended application domains and, second, the software engineering domains we use to model it. Although Cuneiform can be used in other application domains too, its intended application domain is bioinformatics and next-generation sequencing. On the software engineering side we model Cuneiform as a programming language studying its syntax, semantics, and type system. We also model Cuneiform as a distributed system studying the communication patterns among its distributed components.

While Cuneiform's application domain is set through its target audience, we pick the formalisms to model Cuneiform ourselves. We pick reduction semantics to describe Cuneiform's semantics and we pick Petri nets to describe its communication patterns. Although there are other formalisms to model both programming languages and distributed systems we found the combination of reduction semantics and Petri nets a good fit. An exhaustive comparison of candidates is, however, outside the scope of this thesis.

Thus, in this chapter we give a short introduction to each of the research domains Cuneiform touches from both the application side and the software engineering side. We give an overview of Cuneiform's application domains (bioinformatics and next-generation sequencing) in Section 2.4. Section 2.1 introduces scientific workflows as the subject of modeling. Section 2.2 introduces reduction semantics for modeling programming language semantics and Section 2.3 introduces Petri nets for modeling distributed systems.

2.1. Scientific Workflows

Researchers in bioinformatics and next-generation sequencing use scientific workflows to analyze large data sets. A scientific workflow is a description of the data dependencies between self-contained processing steps [245]. A workflow automates these processing steps and makes them reproducible [78]. Herein, each processing step uses specialized tools and libraries that exchange data using specialized formats. To process large data sets a scientific workflow system runs independent processing steps in parallel.

The concrete processing steps and dependencies that unfold while running a scientific workflow form a directed acyclic graph [23, 245]. This graph can be used to visualize a workflow's execution, to track provenance relationships [48, 207], to rerun parts of a workflow [9, 206], or to observe execution statistics to optimize schedules [119, 144, 246]. In addition, the dependency graph illustrates that scientific workflows rely on data dependencies rather than control structures to organize computation.

For this thesis we need a language-oriented definition of the word scientific workflow. Thus, we first define what a *scientific workflow language* is.

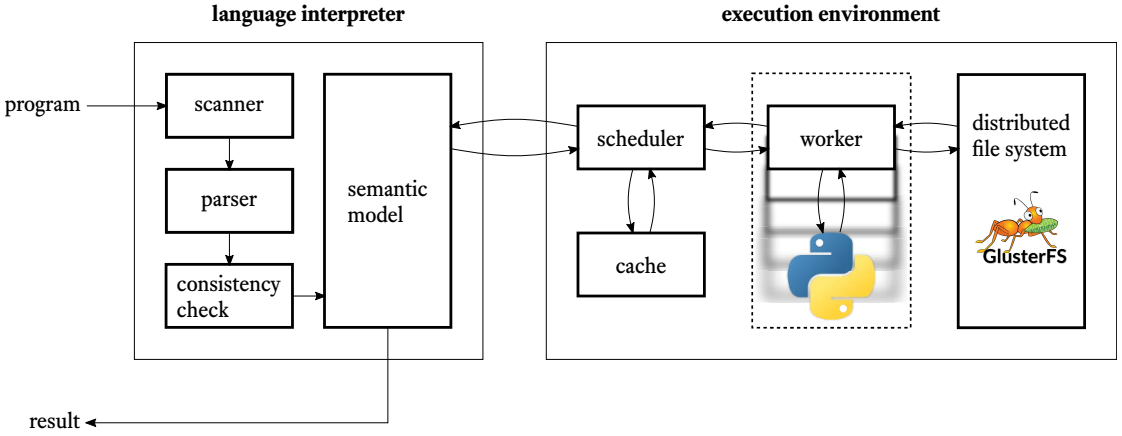


Figure 2.1.: Scientific workflow system components

Definition 1. A scientific workflow language is a language that (i) allows a user to describe computation steps and their data dependencies, (ii) integrates external software, and (iii) runs in an execution environment that parallelizes independent computation steps.

With this definition we can define the term *scientific workflow*:

Definition 2. A scientific workflow is an expression in a scientific workflow language.

Often, scientific workflow languages use a directed acyclic graph to describe data dependencies. In contrast, Cuneiform uses functions and function applications for this purpose. We can, however, construct a directed acyclic graph by observing the concrete function applications a Cuneiform interpreter schedules at runtime.

2.1.1. Scientific Workflow System Components

We regard a scientific workflow system as the composition of distinct components. Together these components form the architecture of a scientific workflow system. The basic components of a scientific workflow system are (i) its front-end, the workflow language interpreter and (ii) its back-end, the distributed execution environment. The workflow language interpreter takes a scientific workflow and extracts independent processing steps from it. The execution environment, in turn, takes independent processing steps and executes them in parallel. When the execution environment learns the result of a processing step, it reports the result back to the interpreter which finds new processing steps. This communication pattern repeats until the interpreter holds the final result of the scientific workflow.

Figure 2.1 shows a detailed schematic of a scientific workflow system and its components. The user interacts with the workflow language interpreter by providing a program in the workflow language and gets a result back. In the following, we discuss the subcomponents of the workflow language interpreter and the distributed execution environment.

Language Interpreter

The front-end of a scientific workflow system is the workflow language interpreter. The interpreter takes a workflow description and returns the workflow result. A workflow language interpreter typically consists of the following subcomponents:

Scanner The scanner performs a lexical analysis of the workflow script, generating a sequence of lexical tokens.

Parser The parser performs a syntactic analysis of the token sequence, generating a workflow model.

Consistency Check Before running the workflow an interpreter performs a static consistency check on the workflow model. Typically, the interpreter screens the workflow model for evident errors like type mismatches, unbound variables, or ambiguous names before running the workflow.

Semantic Model The semantic model reduces the workflow and extracts independent processing steps from it. The interpreter sends each processing step to the distributed execution environment which returns the processing step's result. The semantic model repeats this procedure until it cannot further reduce the workflow.

Each workflow language is the result of a unique perspective on language design. Here, we characterize scientific workflows from the perspective of programming languages. We point out that scientific workflows, like programs, have a syntax and a semantics. To analyze the syntax of a workflow script we take the perspective of a compiler by first treating the symbol sequence as a sequence of tokens in a regular language and then treating the token sequence as a syntax tree in a context-free language [5]. Similarly, to check the consistency of a workflow we take the perspective of types [183], and to reduce the semantic model of a workflow we take the perspective of operational semantics [74].

Workflow languages come in different formats. For instance, graphical languages like KNIME [21] or Taverna [117] often store workflows as serialized binary objects. Here, the serialization mechanisms of a language runtime perform scanning and parsing. E.g., Taverna uses the SCUFL2 format [84] to serialize workflows as serialized Java objects. Other workflow languages use clear-text serialization formats to encode workflows. E.g., Pegasus DAX [36] uses XML while Common Workflow Language [11] uses YAML. These languages rely on an XML or YAML parser library to obtain the workflow model from a script. Another way to perform parsing is to embed an internal language in a host language [77]. E.g., Tez [198] is a fluent API embedded in Java and Nextflow [56] is embedded in Groovy. Here, the runtime of the host language performs scanning and parsing. Often, the type system of the host language also helps to find inconsistencies upfront. Lastly, there are external workflow languages with a dedicated syntax, e.g., Swift [242, 243] or Cuneiform that we study in this thesis. Here, the scanner and the parser are self-contained components. Parser generators like ANTLR [179, 180], Lex/Yacc [142], or Leex/Yecc can generate scanners and parsers from a syntax specification in Backus-Naur Form.

Execution Environment

The back-end of a scientific workflow system is the distributed execution environment. The execution environment takes a processing step and returns its result. Herein, the execution of a processing step is asynchronous, i.e., the interpreter can submit many processing steps without having to wait for the execution environment to return a result. Also, since all processing steps are independent, the execution environment runs them in parallel. A distributed execution environment typically consists of the following components:

Scheduler The scheduler matches a new processing step with a free worker and sends the processing step to that worker. When the worker finishes execution of the processing step it returns a result to the scheduler. The scheduler then relays that result to the language interpreter. If a worker fails, the scheduler must reschedule any processing step which that worker runs.

Cache If the interpreter requests a processing step multiple times the scheduler gets the result from a cache instead of addressing a worker again. Some workflow systems omit the cache or integrate it in the language interpreter instead of the execution environment.

Worker A typical distributed execution environment has many independent workers. Given a processing step, a worker stages in data from a distributed file system, executes the processing step, stages out data to the distributed file system, and sends the result to the scheduler component which relays it to the language interpreter.

Each execution environment provides an interface so that it can receive processing steps from the interpreter. In practice, execution environments and their interfaces vary to fit the workflow languages they back. Some distributed execution environments are specific to variants of only a single workflow. For instance, there are distributed execution environments based on Hadoop [58, 240] specialized to only read mapping workflows like Eoulsan [123], Crossbow [138], CloudAligner [170], DistMap [176], SEAL [185], or CloudBurst [201]. There are also distributed execution environments based on Spark [249] specialized to only alignment processing workflows like ADAM [157] or SparkGA [166]. In contrast, some execution environments bundle with a fixed set of workflows. E.g., Omics Pipe [75] and bcbio-nextgen [95] each bundle with a number of workflows including RNA-seq and variant calling workflows. Other research efforts extend an existing execution environment with domain-specific operators. E.g., both SeqPig [204] and BioPig [171] extend the execution environment Pig [80] that backs Pig Latin [173], an SQL-like query language, with operators for bioinformatics and next-generation sequencing. There are also execution environments specific to exactly one workflow language. E.g., both KNIME and Taverna bundle with a parallel execution environment. Lastly, there are execution environments that support several workflow languages or are language-independent. E.g., HTCondor [146, 214, 215] supports several workflow languages including Nextflow and SDAG[1]. Similarly, there are workflow languages compatible with several distributed

[1] `https://github.com/abdulrahmanazab/sdag`

execution environments. E.g., Nextflow is compatible with several execution environments including HTCondor and Slurm [244].

The research community has also attempted to standardize the interface between language interpreters and execution environments. The DRMAA project [14, 105, 224, 35, 223] is an instance of such a standardization effort.

However, the aforementioned frameworks each solve only a narrow problem class. Scientific workflows allow us to generalize the application domain. They have been used in various research areas, for instance, McRunjob [90] in high-energy physics, Cybershake [40, 51, 91] in geophysics, Montage [20, 53, 199] in astronomy, or CIM-EARTH [69, 70, 71] in energy, climate, and economics modeling. In addition, many scientific workflow systems offer extensions for genomics, for instance, BioKepler [8] for Kepler or KNIME4Bio [145] and next-generation sequencing extensions [118] for KNIME. The possibility of running scientific workflows in distributed environments has been of special interest [125].

Scientific workflows offer (i) a way to define processing steps and to structure their dependencies, (ii) a generic way to integrate foreign tools and libraries such as read mappers, alignment processing tools, or variant callers, and (iii) a way to distribute computation.

Yu and Buyya define a scientific workflow as a collection of tasks processed on grid resources in a specific order [245]. Similarly, Gil et al. characterize workflows as capturing individual data transformations and the mechanisms to distribute these data transformations [83].

Both definitions imply that a workflow, in whatever way it is defined, can be decomposed into tasks transforming data. They also imply that some tasks are independent (otherwise the grid infrastructure would be pointless) while other tasks are dependent (which imposes a specific order). Herein, both definitions leave open how the workflow system accomplishes the decomposition into tasks and the specification of task dependencies. Consequently, a large number of diverse specification languages and systems qualify as scientific workflow systems.

Attempts to come to a comprehensive categorization scheme for scientific workflow systems have been made by various groups [46, 140, 245]. But instead of detailing some categorization scheme or introducing a new one, here, we consider three influential scientific workflow design paradigms manifest in many contemporary workflow systems: Make-like dependency management, dataflow, and functional programming.

2.1.2. Make-Like Workflow Languages

Make [73, 76], is a design reference for many scientific workflow systems. It is a specification language allowing a user to describe computations as data dependencies. Herein, a make script is a collection of Make targets representing the files to be produced. Make targets are specified as triples, defining (i) the name of the output file the target produces, (ii) the names of the files the target depends on, and (iii) a recipe creating the output files of the target given it does not exist yet and all dependencies are met.

Notably, this way of specifying computations as data dependencies aligns closely with

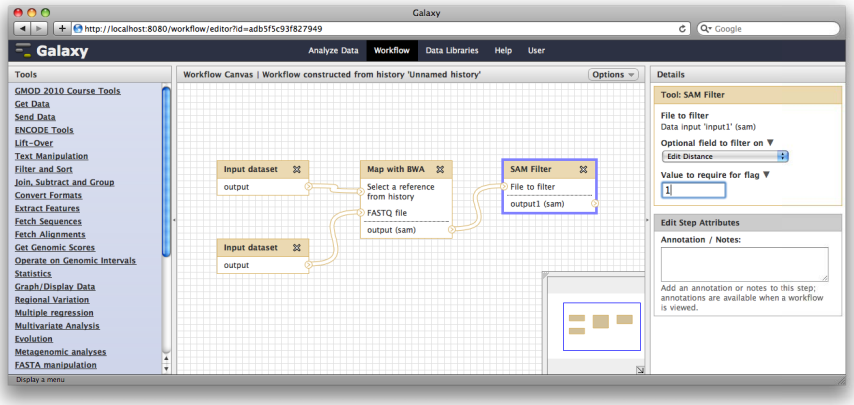


Figure 2.2.: Galaxy workflow. Galaxy represents workflows as a static, cycle-free dependency graph.

our previous definition of scientific workflows. The Make script is made up of targets corresponding to workflow tasks. Furthermore, the (partial) order of workflow tasks is the transitive closure of all target dependencies. In addition, many Make implementations allow to run independent targets in parallel. This correspondence to the definition of scientific workflows and also its maturity constitute Make as an important reference of comparison for scientific workflow systems.

Make's limitation to a single workstation and also its error handling, which is adequate for the time it was conceived but does not meet modern standards, limit its applicability. However, Make targets are consistent, i.e., targets can be made up of other targets, and also composable, i.e., targets are independent of state or targets outside their dependency closure. Both properties contribute to its aptitude to express complex dependency hierarchies.

Making the dependency graph directly accessible is a common design principle in scientific workflow languages. I.e., they regard workflows as a collection of tasks and their dependencies adding few if any additional facilities of abstraction. One prominent example for this is Pegasus DAX [52, 54] which is a specification language in the form of an XML document specifying each task explicitly as a command line tool, its arguments and its input data. Rather than a workflow specification language for manual editing, DAX is intended as an intermediate format produced by more abstract DAX APIs available for Java, Perl, or Python [24, 159].

A workflow specification language that more directly borrows from Make is Snakemake [135, 136]. Snakemake is a Python-based domain-specific language which allows the specification of targets much in the style of Make, however, it runs on distributed compute infrastructures like Sun Grid Engine [81]. Other examples for Make-like workflow systems include Galaxy [82, 86] (see Figure 2.2), Common Workflow Language [11], Rmake [102], BioMake [112], Makeflow [6], Workflow Description Language [2], World-

[2] `https://software.broadinstitute.org/wdl/`

Make [3], Remake [4], or Drake [5].

2.1.3. Dataflow-Based Workflow Languages

Make-like workflow languages require the task dependency graph to be static. It also means that the structure of the dependency graph needs to be cycle-free to terminate. (On the upside, in the absence of cycles, termination is guaranteed.)

Some workflow systems avoid this limitation by adopting a dataflow-based design [121]. A dataflow is a directed graph in which the nodes represent operators and the edges represent communication channels. Herein, a dataflow operation is executed as soon as all input operands are available. This means that dataflows can contain conditional operators, sending data to one part of the dataflow graph but leave out another part. In addition, dataflows can have cycles. Consequently, although dataflows have a static structure, the task dependencies unfold only at runtime and if and how often an operator is applied cannot be inferred without running the dataflow.

Like Make, the dataflow architecture supports parallelism by making operator dependencies explicit. So independent, and thus parallelizable, operators can be directly identified. Also, like Make targets, dataflow operators are stateless, which makes them composable. However, since data dependencies unfold only at runtime the added expressiveness of the dataflow architecture is bought at the expense of the predictability of static dependency graphs.

One scientific workflow system based on the dataflow architecture is Kepler [7, 10, 152] (see Figure 2.3). Also, the Swift parallel scripting language [241], while fostering a functional programming-like notation style, has dataflow semantics providing conditionals and cycles [242]. Also the Groovy-based fluent DSL Nextflow [56] and the Go-based framework SciPipe [6] are dataflow-based workflow systems. Naiad [165] implements a special variant of dataflow called timely dataflow [1].

2.1.4. Functional Programming-Based Workflow Languages

Another way to overcome the limitations of the Make-like static dependency representation is to interpret the targets and their dependencies as chains of function applications in a functional programming setting. So, instead of interpreting targets as processes and data dependencies as communication channels, like we did for the dataflow architecture, we now interpret targets as function applications and data dependencies as links to the operands.

The difference between executing a dataflow and executing a functional program is that the dataflow model specifies a communication structure determining how messages flow from one process to another while the functional programming model specifies how

[3] `http://worldmake.org/`
[4] `https://github.com/richfitz/remake`
[5] `https://github.com/Factual/drake`
[6] `https://github.com/scipipe/scipipe`

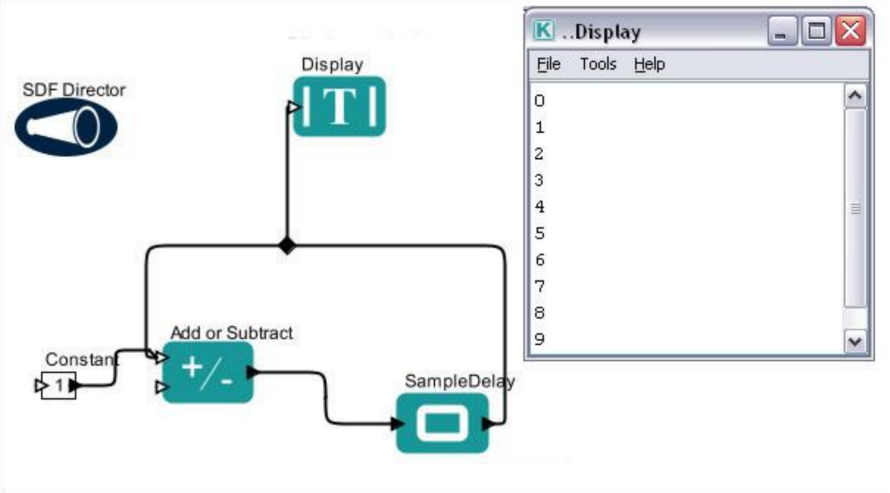


Figure 2.3.: Kepler workflow with cycle. The dataflow-based workflow system represents workflows as directed graphs. In this case, workflows can be cyclic.

expressions are constructed and how they are reduced. Herein, reducing an expression means that the expression is replaced with a simpler version of itself until a value results.

The Church-Rosser property is essential because it allows us to regard sub-expressions as independent. This independence can be used to parallelize and distribute computation.

Applying the functional programming model to the domain of scientific workflows has been proposed by both the Taverna community, who take the approach of formulating Taverna's semantics in terms of a computational lambda calculus [225], and by Peter Kelly, who takes the approach of exploring the lambda calculus a model for scientific workflows [127, 128].

Sometimes the distinction between dataflow and functional programming is blurred. E.g., the Swift parallel scripting language adopts a functional style but its execution an top of the Karajan engine is possible only if the Swift script can be translated to a dataflow graph. Another example is Nextflow which advertises itself to be based on the dataflow programming model but, at the same time, promotes functional composition.[7] Kepler dataflows can be expressed in Haskell to bridge the gap between dataflow and functional programming [151].

So, the dataflow and functional programming model can be treated interchangeably in a number of special cases that exhibit neither cyclic communication paths nor self-reference. In this thesis, however, we distinguish dataflow and functional programming.

2.1.5. Workflows from a Programming Language Perspective

A scientific workflow system allows the user to specify a workflow in some workflow language. Some systems provide a graphical language where the user places nodes and

[7] `https://www.nextflow.io/`

connects them with lines that represent the data dependencies in the workflow. KNIME and Taverna are systems providing such graphical languages. Other systems provide a textual language where the user creates a workflow script that the scientific workflow system consumes. Swift and Nextflow are systems providing such textual languages.

So, all workflow systems have in common that they provide a workflow language as their user interface (textual or graphical) which has a syntax and semantics. The *syntax* of a scientific worklfow language defines a workflow's form. The *semantics* of a scientific workflow language define a workflow's meaning.

Some scientific workflow languages provide an informal specification of their syntax and semantics in the form of a manual. For other languages, like Taverna [225], formal specifications are available in the literature.

We can separate the semantics of a workflow language from distribution aspects like scheduling, caching, data transfer between computers, failure recovery, etc. In this thesis we use that separation to come up with simple semantics that do not touch distribution aspects.

2.1.6. Workflows from a Distribution Perspective

The definition of scientific workflows we gave on Page 19 states that workflows produce independent tasks. Accordingly, we model a workflow system from the perspective of a distributed system by regarding it as a distributed machine that schedules and executes independent tasks.

Modeling scientific workflow execution as a distributed system has been exemplified by the Taverna community by conceding their functional approach in favor of trace semantics, thereby establishing a custom-tailored process calculus for scientific workflows [111, 209, 210, 211]. In contrast, Kepler has been modeled as an actor-oriented system [7, 29]. Gridflow [41], DFL [110], and YAWL [231] are workflow languages modeled as Petri nets.

All these modeling approaches describe the semantics of a workflow language in the framework of a concurrent formalism. In this thesis, however, we seek to separate the semantics of a workflow language from the distribution aspects.

This separation allows us to set up the distributed execution environment of a workflow system as a simple batch execution machine that consumes independent tasks and produces their results. The distributed execution environment is responsible for scheduling, caching, failure recovery, and data transfer. In this thesis we use that separation to come up with a simple distribution model that does not touch any aspects of the workflow structure. The separation between semantics and distribution aspects is reflected in Cuneiform's design separating the Cuneiform interpreter from the distributed execution environment.

2.2. Reduction Semantics

For a workflow language to be general, it needs to compose large workflows from small workflow instances and large data structures from instances of base data types. Also, it

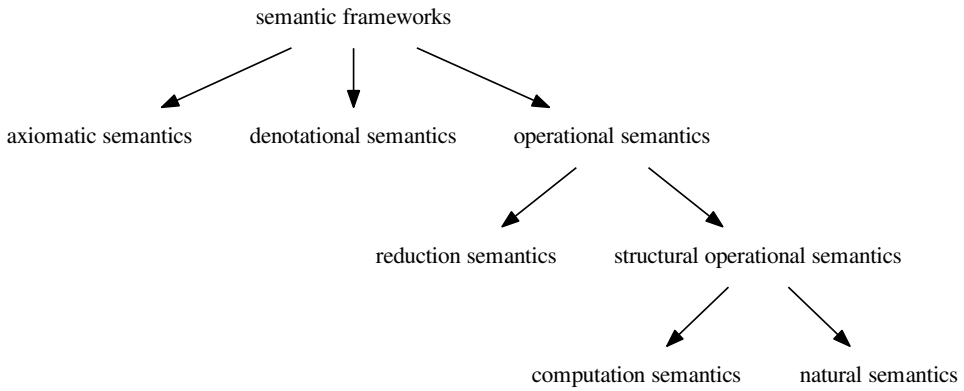


Figure 2.4.: Semantic frameworks and their relationships

needs to be flexible enough to determine the final structure of the workflow at runtime instead of operating on a static workflow graph.

To accomodate this flexibility and expressive power we analyze and design the workflow language Cuneiform the same way that many programming languages are analyzed and designed: by defining it using *operational semantics*.

Scientific workflow languages have much in common with programming languages because they coordinate computation. Therefore, it is natural to judge, analyze and design scientific workflow languages through the perspective of programming languages. Having a defined semantics allows comparison of different language implementations, dependable compilation to and from Cuneiform, and, although this is not part of this thesis, proving the assertion of type safety.

In this thesis we model workflow languages in terms of operational semantics. Other semantic frameworks suitable to specify the semantics of programming languages are denotational semantics or axiomatic semantics. Denotational semantics have been used to specify many programming languages, e.g., Prolog or Concurrent Haskell. In contrast, we use a reduction semantics, a special case of an operational semantics, to specify Cuneiform (see Figure 2.4). The reason for picking an operational semantics is motivated by the purpose of deriving a concrete implementation, as opposed to other possible modeling goals, for instance, establishing the relationship with a particular mathematical framework. Operational semantics specify a programming language (or any formal system) in terms of symbolic manipulations, which are readily implementable in any general purpose programming language. Furthermore, operational semantics are a mature modeling tool that has been used to model many programming languages (e.g., Java and many others). In comparison with structural operational semantics, reduction semantics have the advantage that reduction is defined in two separate steps. First,

Table 2.1.: Explanation of Petri net graphical elements

graphical element	**meaning**
circle	place, passive element, e.g., storage or buffer
rectangle	transition, active element, e.g., process or service
edge from circle	path on which transition consumes data from a place
edge to circle	path on which transition produces data to a place

the definition of the notion of reduction, and second, the extension of this notion to form a standard reduction relation. This two-step approach separates the reduction of a reducible expression (redex) from the identification of the evaluation context in which this redex may appear, which further eases implementation.

We have decided for a reduction semantics although it is not the only viable option in terms of a semantic framework. The semantics of Taverna, both the lambda calculus based semantics and also the later semantics using trace semantics, were presented as natural semantics.

2.3. Petri Nets

We use Petri nets to model communicating systems [182]. Each component of a communicating system may depend on the products of other components, may communicate with other components, or may operate independently of other components. A component may be passive by storing data, or active by performing computation. We can compose components to systems that interact with their environment. Modeling communicating systems with Petri nets helps us to better understand the systems we intend to build and to communicate this understanding [193].

In this thesis we model communicating systems as open place-transition schema nets whose markings consist of arbitrary data. Herein, we subdivide the scientific workflow system we intend to build into several components each of which we model as a net. We call the software component that implements such a net a *service*. Chapter 4 introduces the Petri net models of the Cuneiform interpreter and a corresponding distributed execution environment. In Chapter 5 we introduce a way to directly implement services modeled as Petri nets as Erlang processes.

Petri nets have been used to model various scientific workflow systems. E.g., DFL [110] or YAWL [231]. Alternative formalisms to model scientific workflow systems as distributed systems are actors [2], or process calculi like CSP [186] or the π-calculus [161]. E.g., Kepler has been modeled using actors [29] while Taverna is based on a trace semantics that is inspired by process calculi [111, 211].

2.3.1. Place-Transition Nets with Arbitrary Data

There are several classes of Petri nets. The two most common net classes are condition-event nets (CE-nets) and place-transition nets. Both net classes have in common that

passive elements are represented by circles while active elements are represented by rectangles. In condition-event nets circles represent conditions which can be either enabled or disabled while rectangles represent events. An event occurs when all preconditions but no post-condition is met. In contrast, in place-transition nets circles represent places which are containers for data, while rectangles represent transitions which are operations on data. When a transition fires, it fires in a particular mode indicating which data items are involved in the firing. When the transition fires in a particular mode, the involved data items are consumed in the pre-set of the transition. The data elements a firing produces are a function of the mode. A place in a place-transition net can hold an unbounded number of data tokens.[8]

The annotations of arcs that go from a pre-set place to a transition constrain the possible modes that enable a transition. Additionally, transitions may carry annotations that further constrain when a transition is enabled. The annotations on the net arcs that go from the transition to the post-set determine how the firing of a transition forms the tokens it produces. Herein, any variable used in the annotation of an arc to a post-set place is bound distinctly in the firing mode.

A place-transition net's marking can comprise arbitrary data. When two tokens have the same form then they enable the same transitions in the same modes. This means that two tokens that have the same form are indistinguishable except by observing their respective history.

In this thesis we discuss only place-transition nets with arbitrary data.

2.3.2. Open Nets

Software services interact with their environment. Thus, Petri net models of these services need to provide a way for the environment to interact with the service and vice versa. In this thesis all net models have an interface that separates the service from its environment. The net shares with its environment all elements on the interface. In the Petri net literature we find that all net elements can cross a service's interface: places, transitions, or arcs. If only places appear on the interface we call the net *open*.

A place is a passive state element facilitating asynchronous interaction. I.e., it is up to the environment when to produce or consume tokens on an interface place and conversely it is up to the service to choose when to react to new tokens on an interface place. The message passing mechanisms in Erlang naturally support this style of communication, which is one major reason why we picked Erlang as the implementation language for Cuneiform and its distributed execution environment.

In this thesis we use only open nets to model services. The choice of open nets (interface nets with only places on the boundary) is a convention we pick to make nets more readable. Had we chosen to use transitions as the boundary element, all the above advantages would still apply.[9]

[8] In practice, the computer memory size limits the maximum number of tokens on a place.

[9] In fact, early drafts of Cuneiform's distribution model had transitions as boundary elements. We switched to places only later.

2.3.3. Schema Nets

Often, there can be multiple instantiations of a kind of service. E.g., a worker service that participates in executing a scientific workflow is an instantiation of a worker service template. To model service templates we use schema nets. A concrete net is an instance of such a schema net. A schema net is a Petri net whose annotations consistently contain a variable identifying the net instance. E.g., the tokens passed around in a worker instance all contain an identifier variable distinguishing each worker instance.

In this thesis, we give services that can have multiple instances, like an interpreter service or a worker service, as schema nets. In contrast, we give services that appear only once in a system, like the scheduler service, as concrete nets.

2.3.4. Distributed Runs

A distributed run is a formal notation for scenarios, i.e., a distributed run describes one concrete course of events produced by a system. We use distributed runs to exemplify how a system reacts to a concrete message. A distributed run is an unfolding of a Petri net. I.e., a run is itself a Petri net. In contrast to a Petri net, however, in a run rectangles represent actions, and circles represent single data items that get consumed or produced by actions. Each action is the firing of a transition. The entirety of tokens in the pre-set of an action constitutes the firing mode. All tokens produced by the transition firing appear in the post-set of the action.

In this thesis we use distributed runs to strike a middle ground between UML sequence diagrams, which we use to visualize scenarios, and place-transition nets, which we use to describe the general system. Accordingly, in Chapter 4 we first show a sequence diagram that we translate into a distributed run. The folding of that run gives us a coarse idea of how the corresponding place-transition net looks. Next, we refine this coarse net until we arrive at a collection of net components that exhaustively model the services that constitute a Cuneiform interpreter and a distributed execution environment. We introduce the means to implement these service models in Chapter 5.

Note that distributed runs are technically only valid for condition-event nets. Nevertheless, we apply this technique to place-transition nets. This generalization is valid only if all tokens are distinguishable by their history even if they have the same form. We achieve this by adding a unique identifier to any token. This identifier usually is a natural number much like a row key in a database. In the nets we present we omit this identifier from our net markings to avoid cluttering the notation. Also in our runs, we assume all tokens are distinguishable although this may not be obvious from the notation. Apart from the applicability of distributed runs, distinguishing tokens is of no practical consequence.

2.4. Bioinformatics and Next-Generation Sequencing

Cuneiform's intended application domains are bioinformatics and next-generation sequencing. These application domains drive the design of Cuneiform's language and

system features. Here, we introduce a number of use cases and discuss their terminology outside the context of any concrete workflow language.

To study human evolutionary history and how genetic variation influences the risk of diseases we use genetic sequence information and the insights of molecular genetics [150]. These methods can be applied to study cancer [196], Parkinson's disease [168], Mendelian disorders [57], cardiovascular diseases [172], or diabetes [158, 250].

The carrier of genetic information, the deoxyribonucleic acid (DNA), is made up of sequences of four DNA nucleotides which represent the smallest unit of information in the genome. The four DNA nucleotides are adenine, cytosine, guanine, and thymine. In computers, we represent sequences of DNA nucleotides as strings, made up of an alphabet of four letters. Usually, the characters `A`, `C`, `G`, and `T` represent the four nucleotides respectively. Sometimes, this alphabet is extended, adding the character `N` as a placeholder for an unknown nucleotide.

The genetic information encoded in the DNA determines the structure and function of proteins which are expressed from it. Protein expression in an eukaryotic cell is a process involving three major steps: (i) a DNA-polymerase *transcribes* the coding region of a gene forming the primary transcript (pre-mRNA), (ii) this primary transcript is *processed* transforming it into the mature messenger RNA (mRNA), and (iii) a ribosome *translates* the mature messenger RNA to a polypeptide, an amino acid sequence, which folds and takes up cofactors to form the functional protein.

While the *coding regions* of the DNA serve as a template for the amino-acid sequence underlying a protein, the *non-coding regions* and the overall state of the chromatin in a cell's nucleus also play an important role in gene expression: different packing density of the chromatin and interaction with binding factors at specific DNA binding sites regulate gene expression.

Finally, a cell interacts with its environment relaying information from the cell's environment into the cell nucleus, where gene expression takes place. This allows the cell to react to the state of its surrounding, e.g., to respond to stress, or the availability or scarcity of nutrients, to assume its role in an organism made up of many cells, and to respond to signals from other cells.

In the following, we discuss several study types. We discuss DNA-seq, epigenomics, and transcriptomics. Many other study types exist but an exhaustive discussion of genomics and adjacent research areas is out of the scope of this thesis.

2.4.1. Genetic Variants and DNA-Seq

Genetic variants are differences in the nucleotide sequences that make up an organism's genome in comparison to a given reference. Depending on where in the genome a variant occurs, it may alter the sequence of an expressed protein or change how a gene is regulated or processed after transcription. The detection and interpretation of variants plays an important role in genomics. E.g., researchers compare populations by intersecting their variants [219], they characterize diseases by associating a disease with variants [72, 154], and assess the effectiveness of drugs by associating variants and drug effectiveness [114, 131].

A sequence sample typically consists of many million short reads, each a sequence of about a hundred nucleotides. To call variants we first compare each read to a reference genome and find out the position in the reference that best matches the read sequence. This process is called read alignment. If, in some position, a read contains a nucleotide that deviates from the nucleotide in the reference then this position is a candidate for a variant. Usually, each position in the reference is covered by several reads. If most covering reads agree to differ in a particular position, then this position is likely to be a variant. If, however, only very few reads differ in some position, then these reads likely exhibit only a transient sequencing error. In this way, we can find the variants in a genomic sequence sample [234]. For some study types we use additional knowledge to identify and interpret variants, e.g., we assume the ploidity of the organism or filter out known variants.

Correlating genetic variants with known phenotypic traits, e.g., the presence of a disease or the responsiveness to a drug, allows us to interpret the a variant. Also, a variant can appear inside or outside coding regions. Inside a coding region they are silent, change the protein sequence, or introduce a premature stop codon. Previous studies may have identified a variant as common or rare. A variant may be associated with a predisposition, disease, or drug responsiveness. Thus, in final step of a variant call study we interpret each variant in the context of its associated phenotypical traits. We show an example of a variant calling workflow in Section 6.1.

2.4.2. Epigenetics and Methylation

Epigenetics study heritable, reversible modifications of the DNA [192, 26], e.g., histone packing [162, 227] and DNA methylation [25, 122]. Histones are charged molecules that bind and condense chromatin. The chromatin regions condensed by histones are less accessible to DNA-binding proteins. In contrast, DNA methylation means that methyl-groups attached to the chromatin suppress DNA transcription. We call the proteins that attach methyl-groups to chromatin DNA methyltransferases. Conversely, we call the proteins removing methyl-groups DNA de-methylases. Methylation mainly occurs in cytosine residues where the cytosine is followed by a guanine which we call a CpG-dinucleotide [27]. Besides methylation, researchers have studied many other epigenomic DNA modifications. Here, however, we discuss only methylation as an example for epigenomics.

Researchers use sequencing methods to study DNA methylation. To detect methylated sites researchers treat DNA with a solvent that alters only unmethylated nucleotides. Sequencing the resulting DNA and comparing it to an unmethylated reference reveals methylated positions as variants [101].

2.4.3. DNA-Protein interaction and ChIP-Seq

Many proteins interact with the DNA, e.g., transcription factors which promote or suppress transcription, polymerases which synthesize DNA, nucleases which cleave DNA, or histones which pack chromatin. DNA-binding proteins attach themselves to the DNA

at specific binding sites. Finding these binding sites helps researchers understand the roles of DNA-binding proteins in the cell.

The ChIP-seq analysis is a method to find DNA binding sites of transcription factors [233] and other DNA-binding proteins and tends to replace the microarray-based methods for the same purpose [178]. ChIP, chromatin immunoprecipitation, is a method to select DNA fragments that bind only to specific proteins. In a first step, the researcher cross-links the DNA-binding proteins and the chromatin. Next, the researcher uses a DNA nuclease to cleave the DNA-protein complexes into fragments. Now, the researcher uses an antibody to select only the proteins of interest. This process is called immunoprecipitation. Last, the researcher purifies the DNA fragments which are now ready to be sequenced.

Aligning the sequence reads to a reference genome allows insights in the binding sites of the protein of interest. The aligned sequence reads overlap and cover any position in the genome many times. Since the ChIP procedure selects sequences that the protein of interest binds to, the binding sites of the protein have a high coverage while all other regions are only sporadically covered [251]. This means that we can identify protein binding sites by finding peaks in the coverage of the sequence alignment. We discuss a ChIP-seq workflow in Section 6.4.

2.4.4. Transcriptomics and RNA-Seq

To sequence RNA samples researchers back-transcribe RNA to DNA using a reverse transcriptase. Sequencing messenger RNA allows researchers to gather information about the coding regions of the genome. This allows us to detect not only which genes are expressed but also the amount of genetic material that gets transcribed [237]. This way it is possible to compare the gene expression levels of different cell types or cells under different conditions, e.g., under stress or drug influence. We discuss an RNA-Seq workflow in Section 6.3.

2.4.5. Phylogeny

Phylogeny studies the evolutionary relationships of traits of different species. Researchers visualize such evolutionary relationships as phylogenetic trees. Herein, a phylogenetic tree is constructed so that similar species are grouped close together on the same branch of the phylogenetic tree while species that have diverged early in evolutionary history are on different branches of the tree. Furthermore, the length of the branches signifies to what extent the species has evolved. A long branch represents many evolutionary events while a short branch represents a close relationship. The leaves of the phylogenetic tree represent the concrete species while the intermediate nodes represent common ancestors. A phylogenetic tree is generally unrooted. A common technique to find the root in a phylogenetic tree is to include an *out-group* in the phylogenetic study, i.e., a species that is so unlike the rest of the species that the common ancestor of the out-group and the rest of the phylogenetic tree must be the furthest common ancestor of the species under study. Figure 2.5 shows an example of a phylogenetic tree. It visualizes the evolutionary

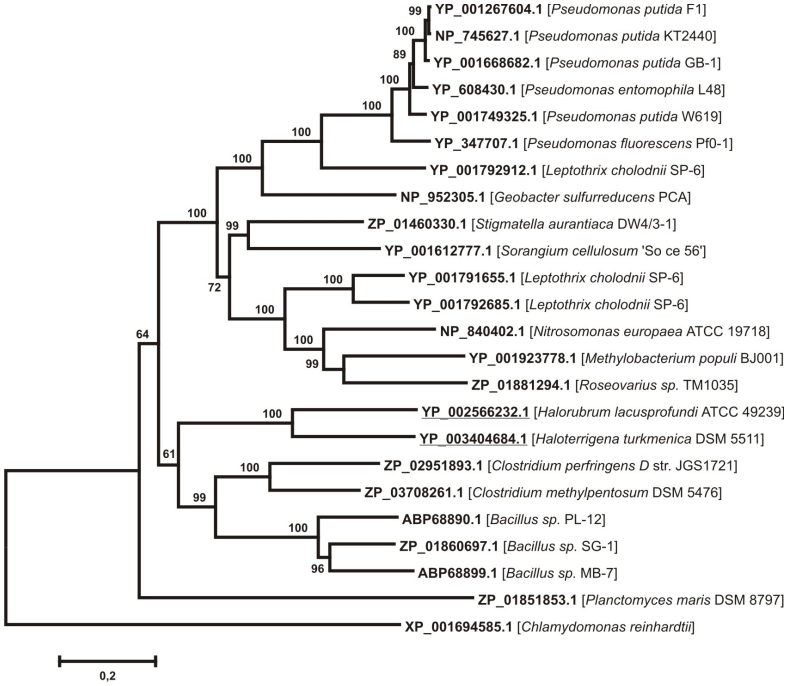


Figure 2.5.: Phylogenetic tree of the six-domain multi-copper blue protein

relationships of various six-domain multi-copper blue proteins [232].

To create a phylogenetic tree we need a distance matrix giving the pairwise distance between all possible combinations of two species. Many algorithms have been conceived to find phylogenetic trees that are both optimal within some quality score like maximum likelihood or Bayesian criterion and robust with respect to missing species or altered distances. Neighbor joining [169] or weighted [208] and unweighted [61, 93, 113] pair group method with arithmetic mean are classical algorithms to find optimal phylogenetic trees. However, contemporary phylogeny software like PhyML [96] use the results of neighbor joining only as initial solutions to improve on. Often these software suites use bootstrapping methods or approximations thereof to get robust solutions.

2.4.6. Sequence Assembly

Sequencers generate many nucleotide sequence reads of limited length. E.g., a human sample may comprise one billion reads, each 100 nucleotides in length. Genome assembly aims to align these reads to one another in way that long contiguous sequences (contigs) result. Tools to perform such de-novo genome assembly are, for instance, SSAKE [238] for DNA short read data or Trinity for RNA-Seq data [89].

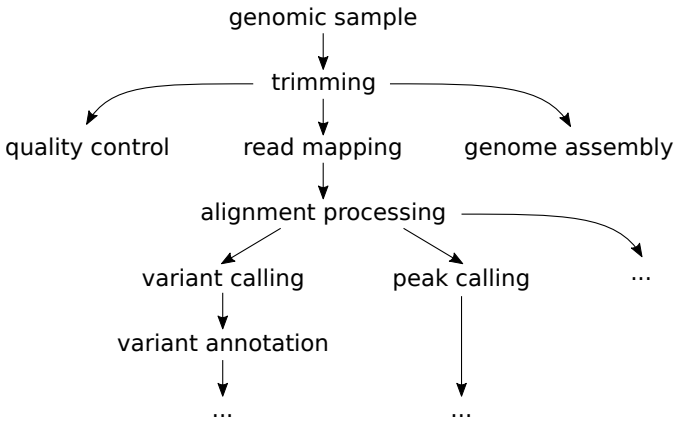


Figure 2.6.: Common genomic processing steps.

2.4.7. Challenges in Genomic Data Analysis

Data analysis in bioinformatics and next-generation sequencing faces several challenges we find only in this area of application. Most importantly, in bioinformatics we deal with heterogeneous software, ranging from libraries in various languages to command line tools, large data sets, easily in the range of several Terabytes, and complex workflows involving, e.g., conditional execution or iteration. Here, we discuss each of these challenges in turn.

Heterogeneous Software

To create a bioinformatics or next-generation sequencing workflow, users need to combine various of tools, algorithms, and programming languages (see Figure 2.6). These tools are also in constant flux since new sequencing methods and study types often imply new requirements towards genomic algorithms. In some cases several software tools may be available to perform the same task. Moreover, despite the effort to standardize important data formats used in bioinformatics, many software tools use custom-tailored, ad-hoc, or semi-structured data formats to exchange data. Therefore, a bioinformatics-focused scientific workflow language needs to take into account that integration of external software needs to be direct and easy and that the workflow system cannot make assumptions about the data being exchanged among tools.

Large Data Sets

In genomics data sets typically range from several Gigabytes for individual samples to several Terabytes for larger studies. For instance, a genomic sample from a human individual with a 30-fold coverage including quality information amounts to ca. 180 GB of data. The most ambitious study so far, the 100000 Genomes Project, is set to gather about 20 Petabytes of sample data.[10]

[10] `https://www.genomicsengland.co.uk/the-100000-genomes-project/`

High-throughput methods for DNA sequencing have improved over the last years so that DNA sequencing today is some order of magnitude cheaper and faster than a decade ago [99, 155, 205]. For many experiment types, multiple sample runs can be sequenced at a time, distinguishing the runs using unique adapter sequences appended to the DNA snippets.

The speed at which data is generated and the potential to run many experiments in parallel has confronted us with the challenge to provide and fully utilize the computational resources analyzing the data so acquired. We face this challenge not only in genomics [181], but also in meta-genomics [203], proteomics [59] and other fields of scientific discovery [18, 108].

Consequently, it is necessary to exploit the parallelism potential in data analysis and to run studies not only across many cores but also across many computers [124, 156, 241]. For that, it is necessary to regard a data analysis platform as a distributed system and to model it accordingly. This holds for scalable programming languages, database systems, workflow systems and big data programming frameworks respectively.

Finally, genomic data analysis steps are often independent. E.g., analyzing one sample is independent from analyzing all other samples in a study up to a certain point in the workflow. Also, quality control is typically independent from the rest of the analysis. This makes large-scale genomics particularly suited for demonstrating Cuneiform's ability to encompass complex studies, to integrate heterogeneous software, and to exploit the potential parallelism in a Cuneiform program.

Complex Workflows

Workloads are typically complex, i.e., the workflow structure is deep, chaining many interdependent analysis steps, and also wide, applying one particular analysis method to a multitude of samples or test conditions.

2.4.8. Further Reading

A profound introduction to the biochemistry aspect of molecular genetics can be found in the Stryer biochemistry book [19] and Griffiths et al. [92] offer a general introduction to genetics but both books only scrape the surface of the computational aspect of genetics which is covered by Lesk [141]. Elliott and Ladomery [68] cover transcriptomics while Krishnarao and Surani [13] provide an introduction to epigenomics. Since DNA sequences are represented in computers as string algorithms play an important role which are covered by Gusfield [100].

3. Cuneiform

A programming language is a textual user interface. It determines the range of things a user can express and how accessible these things are. A programming language should serve a purpose, i.e., it should make expressing problems from a problem domain simple. Typically, a programming language allows a user to build large programs out of small ones. Often, it provides safety features to help find inconsistencies early and minimize turnaround times between writing a program and testing a program.

Regarding scientific workflow systems as a type of programming language allows us to put them in perspective. This perspective brings to light what a worfklow language can express, how practical it is, how composable its constituents are, and how useful its safety features are. By describing it as a programming language we can compare a scientific workflow language to existing programming languages to judge a workflow language's design. In addition, programming language research provides us with tools to specify and reason about languages, e.g., language semantics and type systems.

The connection between scientific workflows and programming languages has been studied by Peter Kelly who, in his 2011 dissertation, uses the lambda calculus as a model for scientific workflows [127, 128]. Taverna's semantics have been modeled as both a typed lambda calculus [225] and a process calculus [111, 209, 210, 211]. Also, Ludäscher et al. suggest that a close relationship connects scientific workflows and pure functional languages [151, 152]. These studies show that we can design scientific workflow languages with the same tools we use to design programming languages.

In this chapter, we study the semantic model of Cuneiform. Concretely, we give the semantics of Cuneiform as reduction semantics in combination with a communication protocol between the Cuneiform interpreter and the distributed execution environment.

Usually, a language's semantics describe the meaning of a program exhaustively. This is not the case here. The semantics we give here describe Cuneiform's abstraction, composition, and consistency features and define a protocol for communication with a distributed execution environment. But only the semantics we give here in combination with the semantics of the execution environment can exhaustively describe the meaning of Cuneiform programs.

By separating the semantics of the interpreter from the semantics of the execution environment we define a Cuneiform workflow system to comprise an interpreter that is concerned with abstraction, composition, and consistency and an execution environment that is concerned with distribution and parallelism. Here we discuss only the Cuneiform interpreter semantics. Separating the language from the distribution eases the construction of Cuneiform because no system component needs to deal with all aspects in combination.[1]

[1]Note that the Term "Cuneiform" refers to the workflow language, which is distinct from the distributed

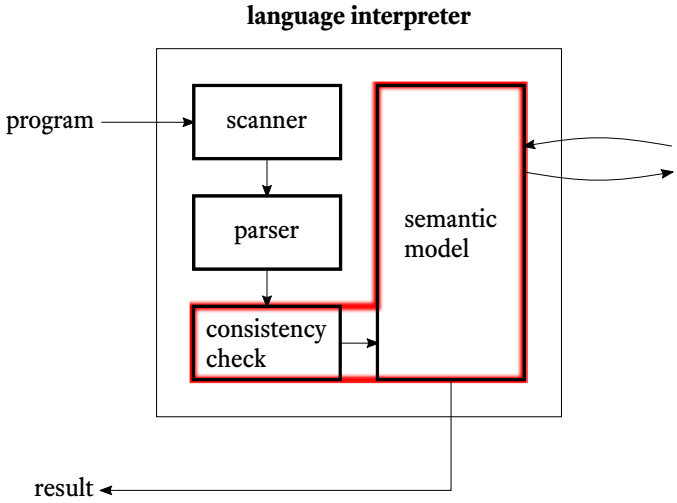


Figure 3.1.: Cuneiform interpreter components. The language semantics specify the consistency check (type check) and the semantic model.

Much in line with the programming language research we define consistency in terms of types and type-check any Cuneiform program prior to execution (see Figure 3.1).

This chapter is structured as follows: First, we give an intuitive overview over Cuneiform's language features by showing and explaining examples in the Quick Tour in Section 3.1. To define Cuneiform's reduction semantics we first give its abstract syntax in Section 3.2. Next, we give a simple type system for Cuneiform as a way to statically check a program's consistency in Section 3.3. Section 3.4 extends the abstract syntax with the syntactic concepts we need to talk about reduction: We distinguish values from general expressions, give a definition of an evaluation context, and give the syntax of a program which gates reduction and communication with the distributed execution environment. Section 3.5 defines the reduction relation in two parts: First, we give a notion of reduction which defines small evaluation steps on reducible expressions. Next we extend the notion of reduction by applying it in an evaluation context resulting in Cuneiform's reduction relation. Lastly, Section 3.7 outlines further reading on the workflows and programming language research.

system that executes independent foreign function applications. The semantics we give in this chapter define Cuneiform almost entirely. Still, technically, the semantics we give here lack the machinery that produces foreign function application results. So, the semantics of Cuneiform are exhaustive only if giving both the semantics of the interpreter and the semantics of the execution environment. We discuss the execution environment in Chapter 4. Note also, that we implemented both: the language interpreter and a suitable distributed execution environment as a contribution for this thesis. We describe the implementation in Chapter 5.

3.1. Quick Tour

With Cuneiform we want to express scientific workflows in bioinformatics and next-generation sequencing. We apply methods from programming language and type theory to come up with a language tailored for these application areas.

From the perspective of scientific workflows, we need to rethink many of the design decisions natural in general-purpose programming languages to express scientific workflows. E.g., in most general-purpose languages users build up software from small, testable units. Libraries are, by default, written in the language itself. In contrast, scientific workflow building blocks often consist of large software suites, that are complex systems in themselves. Also, libraries, by default, are integrated from outside sources. Lastly, many general purpose languages focus on single-core performance. In contrast, scientific workflow systems need to incorporate the potential for parallelism not just in a multi-core computer but in a computer network.

From the perspective of programming languages, we need to rethink the design decisions natural in scientific workflows regarding abstraction and composition. Scientific workflow languages often represent workflows as directed acyclic graphs with nodes as data transformation operations and edges as data dependencies. To extend a scientific workflow the user adds new nodes and connects them to the existing nodes via new edges. However, this representation is limiting if we compare it to the way functional programs represent data flow. In a functional program, a data processing operator is itself a piece of data that can be processed. Also, functional programs allow a user to describe data dependencies in abstract ways that naturally enable iteration, choice, and nesting. Existing scientific workflow languages vary largely in their abstraction features but they usually offer only a subset of the capabilities of functional programming.

There are two major drivers for Cuneiform: (i) the scientific workflow perspective, that focuses on integration and parallelism, and (ii) the functional programming perspective, that focuses on abstraction and composition. In this section we give a light introduction to Cuneiform's language features and show how they may be used in a bioinformatics or next-generation sequencing context.

3.1.1. Automation and Reproducibility

A scientific workflow (i) documents in detail the processing steps leading to a specific result and (ii) automates the processing steps, which a user would otherwise have to apply manually in the right order.

In Cuneiform, we express data dependencies as sequences of function applications. The following Cuneiform program describes some typical data dependencies in a next-generation sequencing workflow.

```
%% input data
let fa : File = 'reference-genome.fa';
let fq : File = 'sample.fq';

%% data dependencies
```

```
let idx : File = bowtie-build( fa = fa );
let alignment : File = bowtie-align( idx = idx, fq = fq );

%% goal
alignment;
```

The program specifies the input data of the workflow: a reference genome and a sample. It also specifies data dependencies: `bowtie-align` produces the alignment while depending on the index and the sample. `bowtie-build`, in turn, produces the index; it depends on the reference genome. Last, the program specifies the goal of the workflow: producing the alignment. The workflow can be run, rerun, and shared among users. It is an exhaustive specification of all processing steps involved in a data analysis pipeline. This way scientific workflows promote understanding, repeating, modifying, and sharing of scientific results.

3.1.2. Integrating External Software

Scientific workflows apply software from external sources. Thus, users need a way to integrate external software into a workflow. This software can come in various forms. E.g., some tools expose a command line interface. Other tools come as R or Python libraries. The user needs to integrate all these software artifacts while hiding their implementation differences.

To integrate a piece of external software in Cuneiform the user defines a foreign function. Calling a foreign function works the same way as calling a native function. The only difference is that a foreign function's body is given as a script in a language other than Cuneiform. This way the user can use different tools and libraries through a consistent interface. The example below shows the definition of a foreign function that integrates the `gunzip` command line tool.

```
def gunzip( gz : File ) -> <file : File>
in Bash *{
  file=unzipped_${gz%.gz}
  gzip -c -d $gz > $file
}*

gunzip( gz = 'my-compressed-file.gz' );
```

Herein, we define the function `gunzip` under the contract that, given that the variable `gz` is bound to an existing file it will bind the variable `file` to a string designating the filename of an existing file which is the result of the function application.

3.1.3. Parallelism

Dependent processing steps need to run in order, in each step transforming their input to be processed by the next step. But *independent* processing steps can run in parallel if compute resources are available. Especially next-generation sequencing workflows process large data sets and typically run for days or weeks. At the same time data

processing steps like sequence alignment are parallelizable. Thus, users expect compute resources to be saturated whenever independent processing steps are available to run.

In Cuneiform all functions are considered deterministic. I.e., whenever a function is applied to some fixed argument, it produces the same result.[2] Consequently, if there is no data dependency between two function applications, the applications are independent and can run in parallel. This way, the Cuneiform interpreter finds and schedules independent function applications. In the following snippet, both applications of `gunzip` are independent and, thus, run in parallel if possible.

```
let a : File = 'a.gz';
let b : File = 'b.gz';

[gunzip( gz = a ),
 gunzip( gz = b ) : File];
```

Here, we define a pair of variables `a` and `b` each holding a file. Next, we define a list with two elements being the unzipped versions of `a` and `b` by calling the function `gunzip` on both variables. Since both applications of `gunzip` are independent Cuneiform runs them in parallel if possible.

3.1.4. Example: Fizz Buzz

Fizz Buzz is often used as an interview question. Here, we show how to write Fizz Buzz in Cuneiform. Imran Ghory states the Fizz Buzz problem as follows:

> Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".[3]

Cuneiform has no dedicated facilities for printing since printing is an unsupported side effect. So, instead of printing, we produce a list of strings. Cuneiform also has no facilities for arithmetic operations, so generating numbers and testing for multiples has to be done in a foreign function. Here, we use Matlab to supplement for arithmetic operations.

```
def range( first : Str, last : Str ) ->
      <number_lst : [Str]>
in Matlab *{
  a = str2num( first );
  b = str2num( last );
  m = a:b;
  c = num2cell( m );
  number_lst = cellfun( @num2str, c, 'UniformOutput', false );
}*
```

[2] We enforce determinism only for native functions. We assume foreign functions to be deterministic too but the user has to enforce it.

[3] `https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/`

```
def is_multiple( number : Str, divisor : Str ) ->
      <is_multiple : Bool>
in Matlab *{
  a = str2num( number );
  b = str2num( divisor );
  is_multiple = mod( a, b ) == 0;
}*

let last : Str = 100;

let <number_lst = number_lst : [Str]> =
  range( first = 1,
         last  = last );

let fizzbuzz_lst : [Str] =
  for x : Str <- number_lst do

    let <is_multiple = f : Bool> =
      is_multiple( number  = x,
                   divisor = 3 );

    let <is_multiple = b : Bool> =
      is_multiple( number  = x,
                   divisor = 5 );

    if ( f and b ) then "FizzBuzz"
    else
      if f then "Fizz"
      else
        if b then "Buzz"
        else
          x
        end
      end
    end

    : Str
  end;

fizzbuzz_lst;
```

The program defines two foreign functions: `range` and `is_multiple`. The `range` function takes two numbers `first` and `last` and creates an ordered list holding each natural number within the interval spanned by both numbers. The `is_multiple` function takes two arguments `number` and `divisor` and determines whether `number` is a multiple of `divisor`. Using these two foreign functions we then use `range` to create a list with the numbers from 1 to 100. Next, we determine for each element of the list whether it is a multiple of 3 or 5 using the function `is_multiple`. In a nested condition we test for

each of the three cases in which a string should be output. If none of the string cases applies the original number is used.

3.1.5. Variable Assignment

We assign a value to a variable and retrieve a variable's content like so:

```
let x : Str =
  "foo";

x;
```

3.1.6. Booleans and Conditions

We branch execution based on conditions using conditional statements. Conditionals are expressions.

```
let x : Str =
  if true
  then
    "bla"
  else
    "blub"
  end;

x;
```

The above variable assignment binds the string `"bla"` to the variable `x`. Then, we query the variable.

3.1.7. Lists

We construct list literals by enumerating their elements in square brackets and declaring the type of the list elements.

```
let xs : [Bool] =
  [true, false, true, true : Bool];

xs;
```

Here, we define the list `xs` whose elements are of type `Bool` giving four Boolean values of which only the second is `false`.

3.1.8. Records and Pattern Matching

A record is a collection of fields that can be accessed via their labels. Literal records can be constructed like so:

```
let r : <a : Str, b : Bool> =
  <a = "blub", b = false>;
```

```
let z : Str =
  ( r|a );

z;
```

We define a record `r` with two fields `a` and `b`, of types `Str` and `Bool` respectively. The field associated with `a` gets the value `"blub"` while the field associated with `b` gets the value `false`. In the last line we access the a field of the record `r`.

Alternatively, we can access record fields via pattern matching:

```
let r : <a : Str, b : Bool> =
  <a = "blub", b = false>;

let <a = z : Str> =
  r;

z;
```

In the second statement we associate the variable `z` with the field a of record `r`. In the last statement we query the content of `z`.

3.1.9. Native Functions

Defining native functions in Cuneiform is done by giving the function name, its signature, and a body expression in curly braces:

```
def id( x : Str ) -> Str {
  x
}

id( x = "bar" );
```

In the first line we define the function `id` which consumes an argument `x` of type `Str` and produces a return value of type `Str`. In the second line, the body expression is just the argument `x`. In the last line we call the function binding the argument `x` to the value `"bar"`.

3.1.10. Foreign Functions

Defining foreign functions is done by giving the function name, its signature, the foreign language name, and the function body in mickey-mouse-eared curly braces.

```
def greet( person : Str ) -> <out : Str> in Bash *{
  out="Hello $person"
}*

greet( person = "Peter" );
```

The first line defines a foreign function greet taking one argument person of type Str and returning a tuple with a single field out of type Str. The foreign function body is

given in Bash code. In the last line we call the foreign function, binding the argument person to the string value "Peter".

3.1.11. Recursion

Many bioinformatics algorithms are iterative. They begin with an initial solution and repeatedly improve that solution until it satisfies a convergence criterion. For example, the K-means clustering we describe in Section 6.6 is an iterative method. It starts with an arbitrary clustering and repeatedly increases the likelihood of that clustering until it cannot improve the solution. For such algorithms it is impossible to tell in advance how many iterations are necessary to meet the convergence criterion. Thus, we need language features that can decide at runtime how often an iterative expression is invoked. In programming languages this unbounded iteration can be achieved by introducing recursion.

In Cuneiform, native function definitions are automatically recursion enabled. I.e., the user can use a function's name in the body of that function. E.g., the following snippet creates an infinite loop:

```
def f() -> Str {
  f()
}

f();
```

The above snippet defines a function `f` which is supposed to return a string. The body calls the function `f` again. As far as the type system is concerned, calling `f` returns a string so the snippet type checks. However, at runtime the snippet diverges because all `f` does is to call itself.

3.1.12. Higher-Order Functions

Often, in bioinformatics and next-generation sequencing applications, one operator can be exchanged with another operator. E.g., a read-mapping operator like Bowtie 2 can, in some cases, be substituted with another read-mapper like BWA, Perm, SHRiMP, bfast, etc. Similarly, a variant caller like VarScan can, in some cases, be substituted with another variant caller like SAMtools, GATK, glftools, or Atlas 2 [147]. Sometimes, researchers apply a list of operators to the same data set to compare the results. At other times, researchers need to pick an operator at runtime.

Thus, a workflow language processes not just data, but the processing operators themselves. In functional programming this pattern is called a higher-order function. A higher order function is a function that takes another function as an argument. In Cuneiform, native functions can take native or foreign functions as an argument. Functions can also appear in compound data structures, forming lists of functions. By creating records that contain functions along with other data, Cuneiform can also mimic an object-oriented style.

The following example shows how a function can be passed as an argument to another function in Cuneiform:

```
def greet( person : Str ) -> <out : Str> in Bash *{
  out="Hello $person"
}*

def apply_person_peter(
  f : Fn ( person : Str ) -> <out : Str> ) -> <out : Str> {
  f( person = "Peter" )
}

apply_person_peter( f = greet );
```

In the above example we define a function `greet` that takes an argument `person` and returns a record with a single field `out`. Next, we define another function `apply_person_peter` which takes an argument `f` which must be a function with the same signature as `greet`. `apply_person_peter` applies the given function binding the `person` argument to the string value `"Peter"`.

When applying Cuneiform in bioinformatics applications, we need higher-order functions only rarely. However, in one instance we defined a workflow doing read mapping and alignment processing with different the read mappers. Since read mappers are structurally similar, they take a sample and a reference index and produce an alignment, we could define the workflow as a function that takes the read mapping function as an argument.

3.1.13. Iterating over Lists

To perform an operation on each element of a list, we iterate using for:

```
let xs : [Bool] =
  [true, false, true, true : Bool];

for x : Bool <- xs do
  not x
  : Bool
end;
```

Here, we define a list of four Booleans and negate each element. The way we use Cuneiform's `for` form here resembles Racket's `for/list` or Common Lisp's `map`.

3.1.14. Iterating Element-Wise

Another way of using Cuneiform's `for` form is to iterate over two or more lists of equal length combining the lists element-wise. In the following example we add two vectors represented as Cuneiform lists.

```
def add( a : Str, b : Str ) -> <c : Str> in Perl *{
  $c = $a+$b;
}*
```

```
let v1 : [Str] =
  [1, 2, 3 : Str];

let v2 : [Str] =
  [4, 5, 6 : Str];

let v3 : [Str] =
  for x1 : Str <- v1,
      x2 : Str <- v2 do
    ( add( a = x1, b = x2 )|c )
    : Str
  end;

v3;
```

First, we define the foreign function `add` which allows us to add two numbers. Next, we define the two vectors `v1` and `v2` as two three-element lists. Last, we compute the vector `v3` by calling the previously defined function `add` element-wise on both vectors. Using Cuneiform's `for` form for element-wise iteration resembles Common Lisp's `mapcar` or Erlang's `zipwith`.

3.1.15. Aggregating Lists

Often in large scale data analysis, researchers need to aggregate the elements of a list. E.g., summing the elements of a list of numbers, concatenating files, or joining several tables.

An aggregation operation typically needs three pieces of information: First, it needs an initial accumulator. Next, it needs the list from which to draw elements. Last, it needs a binary operator that takes the current accumulator value and a list element and produces the next accumulator value.

Cuneiform provides a `fold` form that resembles Racket's `for/fold` or Common Lisp's `reduce`. In the `fold` form's head appear typed bindings for the initial accumulator value and the typed to fold. After the keyword `do` comes a body in which the accumulator and the current list element are bound variables. The body follows an `end` keyword. E.g., the following snippet aggregates a list of numbers by adding its elements.

```
def add( a : Str, b : Str ) -> <c : Str> in Python *{
  c = int( a )+int( b )
}*

let xs : [Str] = [1, 2, 3 : Str];

let sum : Str =
  fold acc : Str = 0, x : Str <- xs do
    ( add( a = acc, b = x )|c )
  end;
```

```
sum;
```

Here, we first define the function add which lets us add two numbers in Python and then the string list xs containing the numbers from one to three. We aggregate the sum of the numbers in xs and store it the result in the variable sum. Lastly, we query the sum variable.

3.2. Abstract Syntax

The first step in modeling Cuneiform as a language is to give its abstract syntax. Herein, Cuneiform represents a scientific workflow as an expression. Since Cuneiform is a typed language, expressions contain type annotations given in a small type language. Here we define the syntax for Cuneiform.

With the abstract syntax, we give here, we can recognize well-formed Cuneiform expressions and give Cuneiform's type rules. However, to give a reduction relation we need to extend the syntax. We call this extension Cuneiform's *dynamic syntax* and define it separately in Section 3.4.

3.2.1. Expression Syntax

Any Cuneiform workflow is given as an *expression*. Expressions compose, i.e., large expressions can be constructed from smaller ones and any way to combine expressions yields an expression again. Expressions can be as simple as a literal value or as complex as subworkflows recursively nesting one another. Cuneiform is based on the lambda calculus and shares many features with functional programming languages.

Expressions fulfill a number of tasks: They allow to define native functions and to call them, to integrate foreign software via foreign function definitions that serve as escape hatches to other languages, to call native functions recursively, to define literals for strings files and Booleans, to combine Boolean expressions with Boolean operators, to compare expressions, to define lists and apply operations on lists, to iterate lists, to

define records and apply operations on records, and to define runtime errors.

$$
\begin{array}{rl}
e ::= & x \\
| & (\lambda\ ([x : T] \ldots)\ (\textbf{ntv}\ e)) \\
| & (\lambda\ ([x : T] \ldots)\ (\textbf{frn}\ x\ T\ l\ s)) \\
| & (\textbf{app}\ e\ ([x = e] \ldots)) \\
| & (\textbf{fix}\ e) \\
| & (\textbf{fut}\ e) \\
| & (\textbf{str}\ s) \\
| & (\textbf{file}\ s) \\
| & \textbf{true} \\
| & \textbf{false} \\
| & (e == e) \\
| & (e \wedge e) \\
| & (e \vee e) \\
| & (\neg\ e) \\
| & (\textbf{isnil}\ e) \\
| & (\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e) \\
| & (\textbf{nil}\ T) \\
| & (\textbf{cons}\ e\ e) \\
| & (\textbf{hd}\ e\ e) \\
| & (\textbf{tl}\ e\ e) \\
| & (e + e) \\
| & (\textbf{for}\ T\ ([x : T \leftarrow e] \ldots)\ \textbf{do}\ e) \\
| & (\textbf{fold}\ [x : T = e]\ [x : T \leftarrow e]\ \textbf{do}\ e) \\
| & (\textbf{rcd}\ ([x = e] \ldots)) \\
| & (\pi\ x\ e) \\
| & (\textbf{error}\ s : T)
\end{array}
\tag{3.1}
$$

We use strings s to designate the content of string literals, file literals, foreign function bodies, and error messages. Herein, a string s is a sequence of characters enclosed in double-quotes.

$$
s ::= \quad \texttt{"..."} \tag{3.2}
$$

Next, we use labels x to designate variable names and record field names. Herein, a label x is any sequence of characters that does not appear as a keyword in the syntax. As keywords we reserve **ntv**, **frn**, **app**, **fix**, etc.

$$
\begin{array}{rl}
x ::= & \texttt{a} \mid \texttt{b} \ldots \\
| & \texttt{aa} \mid \texttt{ab} \ldots \\
| & \texttt{ba} \mid \texttt{bb} \ldots \\
| & \ldots
\end{array}
\tag{3.3}
$$

Lastly, we need to define the foreign languages to recognize. Note that foreign function

names are also keywords and need to be distinct from variable names.

$$
\begin{array}{rl}
l ::= & \text{Bash} \\
| & \text{Erlang} \\
| & \text{Elixir} \\
| & \text{Java} \\
| & \text{Javascript} \\
| & \text{Matlab} \\
| & \text{Octave} \\
| & \text{Perl} \\
| & \text{Python} \\
| & \text{R} \\
| & \text{Racket} \\
| & \ldots
\end{array}
\tag{3.4}
$$

3.2.2. Type Syntax

Type annotations appear in function signatures, lists, iterations, or error expressions. The type system uses these type annotations to determine the consistency of an expression. Cuneiform provides the following types: function types, base data types (strings, files, and Booleans), and compound data types (lists and records).

$$
\begin{array}{rl}
T ::= & \mathsf{Str} \\
| & \mathsf{File} \\
| & \mathsf{Bool} \\
| & (\mathsf{Fn}\ ([x : T] \ldots) \to T) \\
| & (\mathsf{Lst}\ T) \\
| & (\mathsf{Rcd}\ ([x : T] \ldots))
\end{array}
\tag{3.5}
$$

3.3. Type System

Composing large expressions from smaller expressions raises the question if all syntactically possible compositions are meaningful. There are two ways to deal with this question. Firstly, we might ascribe a meaning to as many compositions as possible. This approach has the advantage that any possible expression also has a well-defined meaning. But it has the disadvantage that the meaning of a complex workflow is hard to predict. Consequently, finding the source of a problem in a workflow becomes almost impossible because a problem must be tracked through many nested sub-expressions by the user.

An alternative approach is to limit the meaningful compositions and to check the consistency of workflows in advance. This approach has the advantage that users can easily predict the meaning of a workflow. If a composition has no meaning, an error message is generated before running the workflow, i.e., before wasting precious resources.

3.3.1. Comparability

Some Cuneiform expressions can be compared while others cannot. E.g., Boolean expressions or strings can be compared. Lists or records holding comparable expressions can also be compared. In contrast, we do not compare functions because two functions can mean the same even though they do not look the same. This holds for both native and foreign functions. We also do not compare file names. Since Cuneiform implementations may manipulate filenames to avoid collisions two files with different names may still be not only of equal content but even of equal origin. To discern a priori which expression types can be compared we define the unary comparability relation $\mathbf{c}$:

$$\frac{}{\mathbf{c}\ \ \mathsf{Bool}}\ \text{C-BOOL} \tag{3.6}$$

$$\frac{}{\mathbf{c}\ \ \mathsf{Str}}\ \text{C-STR} \tag{3.7}$$

$$\frac{\mathbf{c}\ \ T_1}{\mathbf{c}\ \ (\mathsf{Lst}\ T_1)}\ \text{C-LST} \tag{3.8}$$

$$\frac{\mathbf{c}\ \ T_i \ldots}{\mathbf{c}\ \ (\mathsf{Rcd}\ ([x_i : T_i] \ldots))}\ \text{C-RCD} \tag{3.9}$$

3.3.2. Type Equivalence

Here, we define equivalence as a binary relation between two types. Herein, two types can be equivalent even if they are not equal: e.g., instances of the types $(\mathsf{Rcd}\ ([a : \mathsf{Str}][b : \mathsf{Bool}]))$ and $(\mathsf{Rcd}\ ([b : \mathsf{Bool}][a : \mathsf{Str}]))$ behave the same in some situations although they are unequal.

$$\frac{}{\mathsf{Str} \simeq \mathsf{Str}}\ \text{Q-STR} \tag{3.10}$$

$$\frac{}{\mathsf{File} \simeq \mathsf{File}}\ \text{Q-FILE} \tag{3.11}$$

$$\frac{}{\mathsf{Bool} \simeq \mathsf{Bool}}\ \text{Q-BOOL} \tag{3.12}$$

$$\frac{T_i \simeq T_j \ldots \qquad T_{\text{ret1}} \simeq T_{\text{ret2}}}{(\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret1}}) \simeq (\mathsf{Fn}\ ([x_j : T_j] \ldots) \to T_{\text{ret2}})}\ \text{Q-FN} \tag{3.13}$$

Q-LST

$$\frac{T_1 \simeq T_2}{(\mathsf{Lst}\ T_1) \simeq (\mathsf{Lst}\ T_2)} \tag{3.14}$$

Q-RCD-BASE

$$(\mathsf{Rcd}\ ()) \simeq (\mathsf{Rcd}\ ()) \tag{3.15}$$

Q-RCD-IND

$$\frac{T_1 \simeq T_2 \qquad (\mathsf{Rcd}\ ([x_i : T_i] \ldots)) \simeq (\mathsf{Rcd}\ ([x_j : T_j] \ldots [x_k : T_k] \ldots))}{(\mathsf{Rcd}\ ([x_1 : T_1][x_i : T_i] \ldots)) \simeq (\mathsf{Rcd}\ ([x_j : T_j] \ldots [x_1 : T_2][x_k : T_k] \ldots))} \tag{3.16}$$

3.3.3. Typing Context

When defining the type rules we will often refer to a *typing context* Γ (also called a type environment). We define a typing context as an association list which consists of pairs each associating a variable name and a type. We use the typing context to approximate the binding of variables in native functions.

$$\Gamma ::= \quad ([x : T] \ldots) \tag{3.17}$$

3.3.4. Type Relation

Functions and Function Applications

First, we give type rules for the part of Cuneiform that constitutes the functional language, i.e., native functions, foreign functions, function applications, and the fixpoint operator.

A variable has the type that appears in the typing context of that variable. If the variable name does not appear in the typing context the variable is unbound. If the variable appears multiple times in the typing context only the innermost binding of the variable determines its type.

T-VAR

$$([x_i : T_i] \ldots [x_1 : T_1][x_j : T_j] \ldots) \vdash x_1 : T_1 \qquad \text{when } \neg\text{member}[x_1, (x_i \ldots)] \tag{3.18}$$

A native function has the type declared in its signature. In addition, the native function's body must have the declared return type when binding all function arguments to their declared type.

T-λ-NTV

$$\frac{([x_j : T_j] \ldots [x_i : T_i] \ldots) \vdash e_{\text{body}} : T_{\text{ret}}}{([x_i : T_i] \ldots) \vdash (\lambda\ ([x_j : T_j] \ldots)\ (\mathbf{ntv}\ e_{\text{body}})) : (\mathsf{Fn}\ ([x_j : T_j] \ldots) \to T_{\text{ret}})} \tag{3.19}$$

A foreign function has the type it declares in its signature.

T-λ-FRN

$$\Gamma \vdash (\lambda\ ([x_i : T_i] \ldots)\ (\mathbf{frn}\ x\ T_{\text{ret}}\ l\ s)) : (\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret}}) \tag{3.20}$$

A function application has the type declared as the return type in the function. In addition, all application arguments must match the argument types declared in the signature of the function.

$$\frac{\Gamma \vdash e_f : (\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret}}) \qquad \Gamma \vdash e_i : T_j \ldots \qquad T_i \simeq T_j \ldots}{\Gamma \vdash (\mathbf{app}\ e_f\ ([x_i = e_i] \ldots)) : T_{\text{ret}}}\ \text{T-APP} \tag{3.21}$$

General Recursion

The fixpoint operator allows to construct recursive functions. The operand function must provide an extra argument with a fitting function type that acts as the function name in the body of the function.

$$\frac{\begin{array}{c}\Gamma \vdash e_f : (\mathsf{Fn}\ ([x_f : (\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret1}})][x_i : T_j] \ldots) \to T_{\text{ret2}}) \\ T_i \simeq T_j \ldots \qquad T_{\text{ret1}} \simeq T_{\text{ret2}}\end{array}}{\Gamma \vdash (\mathbf{fix}\ e_f) : (\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret1}})}\ \text{T-FIX} \tag{3.22}$$

For example

$$\frac{\dfrac{\dfrac{}{([\mathtt{x} : \mathsf{Str}]) \vdash \mathtt{x} : \mathsf{Str}}\ \text{T-VAR}}{() \vdash (\lambda\ ([\mathtt{x} : \mathsf{Str}])\ (\mathbf{ntv}\ \mathtt{x})) : (\mathsf{Fn}\ ([\mathtt{x} : \mathsf{Str}]) \to \mathsf{Str})}\ \text{T-}\lambda\text{-NTV} \qquad \dfrac{() \vdash (\mathbf{str}\ \texttt{"bla"}) : \mathsf{Str}}{\mathsf{Str} \simeq \mathsf{Str}}\ \begin{array}{l}\text{T-STR} \\ \text{Q-STR}\end{array}}{() \vdash (\mathbf{app}\ (\lambda\ ([\mathtt{x} : \mathsf{Str}])\ (\mathbf{ntv}\ \mathtt{x}))\ ([\mathtt{x} = (\mathbf{str}\ \texttt{"bla"})])) : \mathsf{Str}}\ \text{T-APP}$$

String and File Literals

Cuneiform distinguishes *string* and *file* literals. Both are character sequences. However, a string stands only for itself while a file is a character sequence that identifies a data object in the file system. Marking a foreign function argument as a file makes the distributed execution environment stage in the corresponding data object from the distributed file system. Likewise, marking a foreign function's output field as a file makes the distributed execution environment stage out the corresponding data object to the distributed file system.

Cuneiform renames files to avoid name collisions. Consequently, two files with different names may have not only the same content but also the same origin. To avoid this problem when comparing filenames Cuneiform allows to compare only strings. Thus, the type system needs to tell strings and files apart to ensure their consistent use.

$$\frac{}{\Gamma \vdash (\mathbf{str}\ s) : \mathsf{Str}}\ \text{T-STR} \tag{3.23}$$

$$\frac{}{\Gamma \vdash (\mathbf{file}\ s) : \mathsf{File}}\ \text{T-FILE} \tag{3.24}$$

Boolean Expressions

Boolean expressions represent truth values to be used in conditionals or in foreign functions. There are two Boolean literals: **true** and **false**.

$$\frac{}{\Gamma \vdash \textbf{true} : \mathsf{Bool}}\ \textsc{T-true} \tag{3.25}$$

$$\frac{}{\Gamma \vdash \textbf{false} : \mathsf{Bool}}\ \textsc{T-false} \tag{3.26}$$

Comparing two expressions yields a Boolean, under the condition that both expressions are comparable (see Section 3.3.1).

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \mathbf{c}\ \ T_1 \qquad T_1 \simeq T_2}{\Gamma \vdash (e_1 == e_2) : \mathsf{Bool}}\ \textsc{T-cmp} \tag{3.27}$$

Boolean expressions can be combined conjugating, disjugating, or negating them.

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash (e_1 \wedge e_2) : \mathsf{Bool}}\ \textsc{T-conj} \tag{3.28}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash (e_1 \vee e_2) : \mathsf{Bool}}\ \textsc{T-disj} \tag{3.29}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool}}{\Gamma \vdash (\neg\ e_1) : \mathsf{Bool}}\ \textsc{T-neg} \tag{3.30}$$

Lastly, testing if a list is empty yields a Boolean value if the operand has a list type.

$$\frac{\Gamma \vdash e_1 : (\mathsf{Lst}\ T_1)}{\Gamma \vdash (\textbf{isnil}\ e_1) : \mathsf{Bool}}\ \textsc{T-isnil} \tag{3.31}$$

For example

$$((\neg\ \textbf{false}) \wedge (\neg\ \textbf{false}))$$

$$\dfrac{\dfrac{\dfrac{}{() \vdash \textbf{false} : \mathsf{Bool}}\ \textsc{T-false}}{() \vdash (\neg\ \textbf{false}) : \mathsf{Bool}}\ \textsc{T-neg} \qquad \dfrac{\dfrac{}{() \vdash \textbf{false} : \mathsf{Bool}}\ \textsc{T-false}}{() \vdash (\neg\ \textbf{false}) : \mathsf{Bool}}\ \textsc{T-neg}}{() \vdash ((\neg\ \textbf{false}) \wedge (\neg\ \textbf{false})) : \mathsf{Bool}}\ \textsc{T-conj}$$

Conditionals

Conditionals allow to make decisions at runtime. Herein, the expression in condition position needs to be a Boolean. Furthermore, the expressions in then- and else-position need to be of the same type.

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : T_l \qquad \Gamma \vdash e_3 : T_r \qquad T_l \simeq T_r}{\Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) : T_l}\ \text{T-IF} \tag{3.32}$$

For example, the following conditional expression has a negation in condition position and Boolean expressions in then- and else positions.

$$(\mathbf{if}\ (\neg\ \mathbf{true})\ \mathbf{then\ false\ else}\ (\neg\ \mathbf{false}))$$

We type it using the rules T-IF, T-NEG, T-TRUE, and T-FALSE.

$$\dfrac{\text{T-FALSE}\ \dfrac{}{() \vdash \mathbf{false} : \mathsf{Bool}} \qquad \dfrac{\dfrac{\dfrac{\text{T-TRUE}}{() \vdash \mathbf{true} : \mathsf{Bool}}\ \text{T-NEG}}{() \vdash (\neg\ \mathbf{true}) : \mathsf{Bool}} \qquad \dfrac{\text{T-FALSE}}{() \vdash \mathbf{false} : \mathsf{Bool}}\ \text{T-NEG}}{() \vdash (\neg\ \mathbf{true}) : \mathsf{Bool}} \qquad \dfrac{\text{Q-BOOL}}{\mathsf{Bool} \simeq \mathsf{Bool}}}{() \vdash (\mathbf{if}\ (\neg\ \mathbf{true})\ \mathbf{then\ false\ else}\ (\neg\ \mathbf{false})) : \mathsf{Bool}}\ \text{T-IF}$$

Lists

Lists are one of the two compound data types Cuneiform provides. Cuneiform provides two list constructors: nil and cons. The nil constructor carries a type annotation to uniquely identify the list type.

$$\frac{}{\Gamma \vdash (\mathbf{nil}\ T_1) : (\mathsf{Lst}\ T_1)}\ \text{T-NIL} \tag{3.33}$$

The cons constructor carries a head and tail expression. Both the head and the tail must have the corresponding type.

$$\frac{\Gamma \vdash e_1 : T_l \qquad \Gamma \vdash e_2 : (\mathsf{Lst}\ T_r) \qquad T_l \simeq T_r}{\Gamma \vdash (\mathbf{cons}\ e_1\ e_2) : (\mathsf{Lst}\ T_r)}\ \text{T-CONS} \tag{3.34}$$

Analogous to the list constructors Cuneiform provides two list accessors: hd and tl. The hd operator extracts the head of a list operand while the tl operator extracts the tail. If the list operand is the empty list both operators reduce to their default operands. Thus, the the list operand and the default operand must have the corresponding types.

$$\frac{\Gamma \vdash e_1 : (\mathsf{Lst}\ T_l) \qquad \Gamma \vdash e_2 : T_r \qquad T_l \simeq T_r}{\Gamma \vdash (\mathbf{hd}\ e_1\ e_2) : T_l}\ \text{T-HD} \tag{3.35}$$

$$\frac{\Gamma \vdash e_1 : (\mathsf{Lst}\ T_l) \qquad \Gamma \vdash e_2 : (\mathsf{Lst}\ T_r) \qquad T_l \simeq T_r}{\Gamma \vdash (\mathbf{tl}\ e_1\ e_2) : (\mathsf{Lst}\ T_l)}\ \text{T-TL} \tag{3.36}$$

For example, the following expression extracts the head of a two-element string list.

$$(\mathbf{hd}\ (\mathbf{cons}\ (\mathbf{str}\ \texttt{"1"})\ (\mathbf{cons}\ (\mathbf{str}\ \texttt{"2"})\ (\mathbf{nil}\ \mathsf{Str})))\ (\mathbf{error}\ \texttt{"empty list"} : \mathsf{Str}))$$

We can type this expression by applying the rules T-HD, T-NIL, T-CONS, T-STR, and T-ERROR.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{() \vdash (\mathbf{str}\ \texttt{"2"}) : \mathsf{Str}}\text{T-STR} \quad \dfrac{}{() \vdash (\mathbf{nil}\ \mathsf{Str}) : (\mathsf{Lst}\ \mathsf{Str})}\text{T-NIL} \quad \dfrac{}{\mathsf{Str} \simeq \mathsf{Str}}\text{Q-STR}}{() \vdash (\mathbf{cons}\ (\mathbf{str}\ \texttt{"2"})\ (\mathbf{nil}\ \mathsf{Str})) : (\mathsf{Lst}\ \mathsf{Str})}\text{T-CONS} \quad \dfrac{}{\mathsf{Str} \simeq \mathsf{Str}}\text{Q-STR}}{() \vdash (\mathbf{cons}\ (\mathbf{str}\ \texttt{"1"})\ (\mathbf{cons}\ (\mathbf{str}\ \texttt{"2"})\ (\mathbf{nil}\ \mathsf{Str}))) : (\mathsf{Lst}\ \mathsf{Str})}\text{T-CONS} \quad \dfrac{}{() \vdash (\mathbf{error}\ \texttt{"empty list"} : \mathsf{Str}) : \mathsf{Str}}\text{T-ERROR} \quad \dfrac{}{\mathsf{Str} \simeq \mathsf{Str}}\text{Q-STR}}{() \vdash (\mathbf{hd}\ (\mathbf{cons}\ (\mathbf{str}\ \texttt{"1"})\ (\mathbf{cons}\ (\mathbf{str}\ \texttt{"2"})\ (\mathbf{nil}\ \mathsf{Str})))\ (\mathbf{error}\ \texttt{"empty list"} : \mathsf{Str})) : \mathsf{Str}}\text{T-HD}$$

We can append two lists if their elements have the same type.

$$\frac{\Gamma \vdash e_1 : (\mathsf{Lst}\ T_l) \qquad \Gamma \vdash e_2 : (\mathsf{Lst}\ T_r) \qquad T_l \simeq T_r}{\Gamma \vdash (e_1 + e_2) : (\mathsf{Lst}\ T_l)}\ \text{T-APPEND} \tag{3.37}$$

Cuneiform provides two ways to iterate over lists: for-iteration and fold-iteration. The for-iteration iterates element-wise over several lists by binding the head of each list to a variable. Then, Cuneiform evaluates the iteration body with that binding. In this, for-iteration is similar to Racket's `for/list`, Common Lisp's `mapcar`, or Erlang's `zipwith`. Herein, the common `map` is a special case of for-iteration where we iterate over only one list.

For Iteration

To type a for-iteration, we expect each variable binding's type to match the type of the list to iterate. Furthermore, under the assumption that the variable bindings have the declared type, we expect the iteration body to have the declared body type.

$$\frac{\begin{array}{c}([x_i : T_i] \ldots) \vdash e_j : (\mathsf{Lst}\ T_k) \ldots \\ T_j \simeq T_k \qquad ([x_j : T_j] \ldots [x_i : T_i] \ldots) \vdash e_{\text{body}} : T_{\text{body}} \qquad T_{\text{ret}} \simeq T_{\text{body}}\end{array}}{([x_i : T_i] \ldots) \vdash (\mathbf{for}\ T_{\text{ret}}\ ([x_j : T_j \leftarrow e_j] \ldots)\ \mathbf{do}\ e_{\text{body}}) : (\mathsf{Lst}\ T_{\text{ret}})}\ \text{T-FOR} \tag{3.38}$$

The fold-iteration iterates over a list while updating an accumulator for each element in the list. A fold-iteration binds an accumulator variable to an initial accumulator expression, binds an iterator variable to a list to iterate, and provides a body expression.

Fold Iteration

To type a fold-iteration, we expect the initial accumulator expression to be of the declared type. Furthermore, we expect the iterator variable's declared type to match the type of the list to iterate. Lastly, assuming that accumulator and iterator have the declared type, we expect the body expression to have the same type as the accumulator.

$$\text{T-FOLD}\quad \frac{\begin{array}{c}([x_i : T_i]\ldots) \vdash e_{\text{acc}} : T_{\text{acc2}} \\ ([x_i : T_i]\ldots) \vdash e_{\text{lst}} : (\mathsf{Lst}\ T_{\text{lst2}}) \qquad ([x_{\text{acc}} : T_{\text{acc1}}][x_{\text{lst}} : T_{\text{lst1}}][x_i : T_i]\ldots) \vdash e_{\text{body}} : T_{\text{body}} \\ T_{\text{acc1}} \simeq T_{\text{acc2}} \qquad T_{\text{acc1}} \simeq T_{\text{body}} \qquad T_{\text{lst1}} \simeq T_{\text{lst2}}\end{array}}{([x_i : T_i]\ldots) \vdash (\textbf{fold}\ [x_{\text{acc}} : T_{\text{acc1}} = e_{\text{acc}}]\ [x_{\text{lst}} : T_{\text{lst1}} \leftarrow e_{\text{lst}}]\ \textbf{do}\ e_{\text{body}}) : T_{\text{acc1}}} \tag{3.39}$$

Records and Projection

Besides lists, Cuneiform provides *records* as a compound data type. A record comprises labeled fields, each holding an expression. The type of a record is a record type. A record's type associates each label with a type.

$$\frac{\Gamma \vdash e_i : T_i \ldots}{\Gamma \vdash (\textbf{rcd}\ ([x_i = e_i]\ldots)) : (\mathsf{Rcd}\ ([e_i : T_i]\ldots))}\ \text{T-RCD} \tag{3.40}$$

To access a record we use record *projection*. A projection selects a record field identified by its label. The type of the projection is the type of the record field that it selects.

$$\frac{\Gamma \vdash e_{\text{rcd}} : (\mathsf{Rcd}\ ([x_i : T_i]\ldots[x_1 : T_1][x_j : T_j]\ldots))}{\Gamma \vdash (\pi\ x_1\ e_{\text{rcd}}) : T_1}\ \text{T-}\pi \tag{3.41}$$

For example, the following expression projects the field labeled `x` from a record with two fields.

$$(\pi\ b\ (\textbf{rcd}\ ([\mathtt{a} = (\textbf{str}\ \texttt{"ok"})][\mathtt{b} = \textbf{true}])))$$

We type it using the rules T-π, T-RCD, T-STR, and T-TRUE.

$$\dfrac{\dfrac{\dfrac{}{() \vdash (\textbf{str}\ \texttt{"ok"}) : \mathsf{Str}}\ \text{T-STR} \qquad \dfrac{}{() \vdash \textbf{true} : \mathsf{Bool}}\ \text{T-TRUE}}{() \vdash (\textbf{rcd}\ ([\mathtt{a} = (\textbf{str}\ \texttt{"ok"})][\mathtt{b} = \textbf{true}])) : (\mathsf{Rcd}\ ([\mathtt{a} : \mathsf{Str}][\mathtt{b} : \mathsf{Bool}]))}\ \text{T-RCD}}{() \vdash (\pi\ b\ (\textbf{rcd}\ ([\mathtt{a} = (\textbf{str}\ \texttt{"ok"})][\mathtt{b} = \textbf{true}]))) : \mathsf{Bool}}\ \text{T-}\pi$$

Errors

A user-defined error breaks evaluation at runtime. Thus, user-defined errors need to just pass type checking. A user-defined error has the type it declares to have.

$$\frac{}{\Gamma \vdash (\textbf{error}\ s : T_1) : T_1}\ \text{T-ERROR} \tag{3.42}$$

For example, the following expression has an error as the operand of a negation.

$$(\neg\ (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool}))$$

We type it using the rules T-NEG and T-ERROR.

$$\frac{\dfrac{}{() \vdash (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool}) : \mathsf{Bool}}\ \text{T-ERROR}}{() \vdash (\neg\ (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool})) : \mathsf{Bool}}\ \text{T-NEG}$$

In another example, an expression contains an error nested in a Boolean disjugation:

$$(\textbf{true} \vee (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool}))$$

We type it using the rules T-DISJ, T-TRUE, and T-ERROR.

$$\frac{\dfrac{}{() \vdash \textbf{true} : \mathsf{Bool}}\ \text{T-TRUE} \qquad \dfrac{}{() \vdash (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool}) : \mathsf{Bool}}\ \text{T-ERROR}}{() \vdash (\textbf{true} \vee (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool})) : \mathsf{Bool}}\ \text{T-DISJ}$$

3.4. Dynamic Syntax

To formulate a reduction relation for Cuneiform in terms of reduction semantics [74] we need to extend the abstract syntax given in Section 3.2 to formulate the notion of a *value* and a *reduction context*. Briefly, a value is an expression that cannot be reduced and is not an error. An evaluation context is a context in which a reducible expression may appear. We use the abstract syntax, values, and reduction contexts to define Cuneiform's reduction relation in Section 3.5.

3.4.1. Values

A *value* is a non-error expression that cannot be further evaluated. We need to distinguish values from other expressions to define the reduction relation in Section 3.5.2. In the reduction relation we send a foreign function application to the execution environment only if all its arguments are values, i.e., we use a Call-by-Value evaluation strategy

to evaluate foreign function applications. Thus, we need to define a value's form.

$$
\begin{array}{rl}
v ::= & (\lambda\ ([x : T] \ldots)\ (\textbf{ntv}\ e)) \\
| & (\lambda\ ([x : T] \ldots)\ (\textbf{frn}\ x\ T\ l\ s)) \\
| & (\textbf{str}\ s) \\
| & (\textbf{file}\ s) \\
| & \textbf{true} \\
| & \textbf{false} \\
| & (\textbf{nil}\ T) \\
| & (\textbf{cons}\ v\ v) \\
| & (\textbf{rcd}\ ([x = v] \ldots))
\end{array}
\tag{3.43}
$$

3.4.2. Evaluation Contexts

Reduction semantics split an evaluation step into two phases: In the first we look for a sub-expression that is a reducible expression. This sub-expression is embedded in a surrounding expression, which we call the *evaluation context*. In the second we apply the notion of reduction to reduce this expression within the evaluation context.

Here, we define the form of an evaluation context. The evaluation context determines where we can look for reducible expressions. Just like the reduction relations of other programming languages a Cuneiform interpreter looks for reducible expressions in all possible locations. E.g., we do not traverse native function bodies. We also do not traverse the then- and else-blocks of conditionals. Lastly, we do not traverse the body expressions of for- and fold-iterations. However, unlike the reduction relations of other programming languages, the Cuneiform interpreter does not look for reducible expressions deterministically in the left-most outer-most position. Instead, it allows us to find reducible expressions in many locations non-deterministically. So, it can peek ahead and evaluate reducible expressions even if the left-most outer-most position is stalled by a pending foreign function application.

Cuneiform differs from other languages also in that it mixes two evaluation strategies: it uses a Call-by-Value strategy for foreign function applications and a Call-by-Name strategy for native function applications. Thus, we allow an evaluation context in an

application argument position only if the function position holds a foreign function.

$$
\begin{array}{rl}
E ::= & [] \\
| & (\textbf{app}\ E\ ([x = e] \ldots)) \\
| & (\textbf{app}\ (\lambda\ ([x : T] \ldots)\ (\textbf{frn}\ x\ T\ l\ s))\ ([x = e] \ldots [x = E][x = e] \ldots)) \\
| & (\textbf{fix}\ E) \\
| & (E == e) \\
| & (e == E) \\
| & (E \wedge e) \\
| & (e \wedge E) \\
| & (E \vee e) \\
| & (e \vee E) \\
| & (\neg\ E) \\
| & (\textbf{isnil}\ E) \\
| & (\textbf{if}\ E\ \textbf{then}\ e\ \textbf{else}\ e) \\
| & (\textbf{cons}\ E\ e) \\
| & (\textbf{cons}\ e\ E) \\
| & (\textbf{hd}\ E\ e) \\
| & (\textbf{tl}\ E\ e) \\
| & (E + e) \\
| & (e + E) \\
| & (\textbf{for}\ T\ ([x : T \leftarrow e] \ldots [x : T \leftarrow E][x : T \leftarrow e] \ldots)\ \textbf{do}\ e) \\
| & (\textbf{fold}\ [x : T = e]\ [x : T \leftarrow E]\ \textbf{do}\ e) \\
| & (\textbf{rcd}\ ([x = e] \ldots [x = E][x = e] \ldots)) \\
| & (\pi\ x\ E)
\end{array}
\tag{3.44}
$$

3.4.3. Programs

The Cuneiform interpreter only evaluates a scaffold of the computation specified by a Cuneiform expression. A distributed execution environment that hosts many worker processes performs the remaining part of the computation. To exchange messages with the execution environment a Cuneiform program mirrors the communication structure between the interpreter and the execution environment.

This communication structure involves three activities: (i) the interpreter requests the execution environment to execute a foreign function application, (ii) the execution environment replies to the interpreter with the result of an execution in the form of a foreign function application-value pair. (iii) the interpreter makes progress on evaluating the Cuneiform expression.

Accordingly, a Cuneiform program is three-tuple storing an *out-box* that holds foreign function applications to be sent to the execution environment, an *in-box* that holds foreign function application-value pairs just received from the execution environment, and a *control string* that is the current state of the expression to evaluate.

$$
p ::= \quad ((e \ldots)\ ([e\ e] \ldots)\ e) \tag{3.45}
$$

When reduction begins, the interpreter *loads* a Cuneiform expression e_1 by checking its type and, if usccessful, inserting it in the control string position of a program with an empty out-box and an empty in-box.

$$(() \; () \; e_1)$$

Evaluation stops when the program's control string is either a value or an error. Then, the interpreter *unloads* the control string and passes it to the user, disposing of both the out-box and the in-box.

3.5. Reduction

In Section 3.4 we extend Cuneiform's static syntax with the syntactic categories needed to define reduction. We define values, evaluation contexts, and programs. Here, we define the reduction rules as a consistent relation on programs. I.e., reducing a program always yields another program. Here, we use the syntax of values to determine when a foreign function application is ready for scheduling and when evaluation has finished. We use the syntax of evaluation contexts to find reducible expressions inside the control string of a program.

We adopt the style of reduction semantics to define Cuneiform's reduction rules. I.e., we separate the definition of reduction in three parts: First, we define a notion of reduction **n**, then we extend the notion of reduction by applying it in an evaluation context which gives us a first version of the reduction relation. Lastly, we extend the reduction relation with rules for dealing with errors and for communicating with the distributed execution environment. This way we give a small-step reduction relation $\longrightarrow$ for Cuneiform. To reduce a Cuneiform program p_0 we apply the small-step reduction relation until a program p^* results whose control string is either a value or an error expression.

$$p_0 \longrightarrow p_1 \longrightarrow \ldots \longrightarrow p^*$$

3.5.1. Notion of Reduction

Here, we give the *notion of reduction*. The notion of reduction is a relation between a reducible expression and its reduction result.

Functions and Function Application

Function application is handled inductively with in two cases: a base case and an inductive case. The base case states that applying a function with no arguments reduces to the function body.

$$\text{E-}\beta\text{-BASE}$$
$$(\mathbf{app} \; (\lambda \; () \; (\mathbf{ntv} \; e_{\text{body}})) \; ()) \;\; \mathbf{n} \;\; e_{\text{body}} \tag{3.46}$$

The inductive case states that applying a function with at least one argument substitutes the first argument into the function body and disposing of that argument from both the

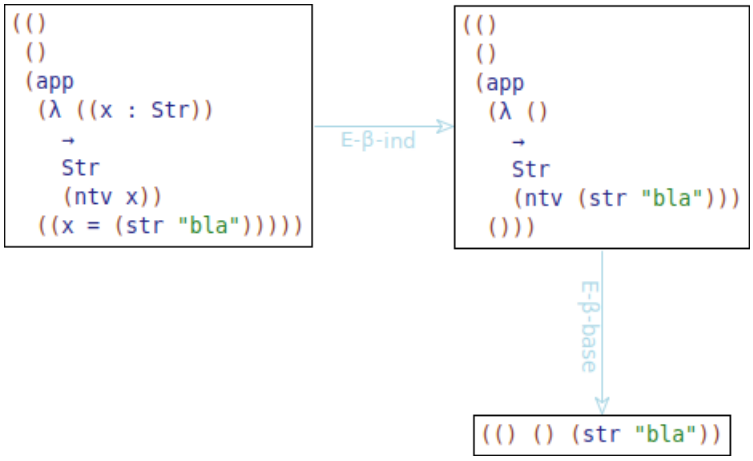


Figure 3.2.: Reduction trace of an application of the identity function on strings. First, we substitute the argument into the function body, then we replace the function application with the function body.

function definition and the application's argument binding list.

$$\begin{array}{l}\text{E-}\beta\text{-IND}\\ (\mathbf{app}\ (\lambda\ ([x_1 : T_1][x_i : T_i]\ldots)\ (\mathbf{ntv}\ e_{\text{body}}))\ ([x_1 = e_1][x_j = e_j]\ldots))\\ \quad \mathbf{n}\ \ (\mathbf{app}\ (\lambda\ ([x_i : T_i]\ldots)\ (\mathbf{ntv}\ e_{\text{body}}[x_1 \leftarrow e_1]))\ ([x_j = e_j]\ldots))\end{array} \tag{3.47}$$

For example, the following expression applies a function taking a single argument `x` to a string literal. The function returns just that string literal.

$$(\mathbf{app}\ (\lambda\ ([\mathtt{x} : \mathsf{Str}])\ (\mathbf{ntv}\ \mathtt{x}))\ ([\mathtt{x} = (\mathbf{str}\ \mathtt{"bla"})]))$$

The interpreter evaluates this expression by substituting each occurrence of the variable `x` with the string expression in the argument. Also, the argument is removed from, both, the argument list of the lambda-expression and the argument binding list of the function application. In a second step, since the function takes no further arguments, the interpreter replaces the function application with the function body inside the evaluation context (see Figure 3.2).

General Recursion

Function definitions can be recursive by using the fixpoint operator. Herein, the operand of the fixpoint operator must be a function with an extra first argument that identifies the function's name. Applying the fixpoint operator removes that argument in the function definition and binds each occurrence of the function name variable to the original fixpoint operator.

The fixpoint operator cannot be defined as a derived form in Cuneiform itself because there is no way of typing such a derived form fixpoint operator in the simple type system

we give in Section 3.3.

E-FIX

$$(\textbf{fix}\ e_1)\ \ \textbf{n}\ \ (\lambda\ ([x_i : T_i] \ldots)\ (\textbf{ntv}\ e'_{\text{body}}))$$
$$\text{when } e_1 = (\lambda\ ([x_f : (\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret}})][x_i : T_i] \ldots)\ (\textbf{ntv}\ e_{\text{body}})) \tag{3.48}$$
$$e'_{\text{body}} = (\textbf{app}\ (\lambda\ ([x_f : (\mathsf{Fn}\ ([x_i : T_i] \ldots) \to T_{\text{ret}})])\ (\textbf{ntv}\ e_{\text{body}}))\ ([x_f = (\textbf{fix}\ e_1)]))$$

Boolean Expressions

One kind of Boolean expression is a comparison. A comparison can be true or false. Generally, the interpreter evaluates both operands of a comparison and then checks if both operands have the same form. Comparison is always deep, i.e., all sub-expressions of the left operand need to be equal to all sub-expressions of the right operand. If a record is involved in comparison, we do not require the record fields to be in the same order. Cuneiform does not directly allow us to compare files. However, it is straight-forward to create a comparison operator as a foreign function.

E-CMP-SEQ

$$((\textbf{str}\ s_1) == (\textbf{str}\ s_1))\ \ \textbf{n}\ \ \textbf{true} \tag{3.49}$$

E-CMP-SNEQ

$$((\textbf{str}\ s_1) == (\textbf{str}\ s_2))\ \ \textbf{n}\ \ \textbf{false} \qquad \text{when } s_1 \neq s_2 \tag{3.50}$$

E-CMP-TT

$$(\textbf{true} == \textbf{true})\ \ \textbf{n}\ \ \textbf{true} \tag{3.51}$$

E-CMP-TF

$$(\textbf{true} == \textbf{false})\ \ \textbf{n}\ \ \textbf{false} \tag{3.52}$$

E-CMP-FT

$$(\textbf{false} == \textbf{true})\ \ \textbf{n}\ \ \textbf{false} \tag{3.53}$$

E-CMP-FF

$$(\textbf{false} == \textbf{false})\ \ \textbf{n}\ \ \textbf{true} \tag{3.54}$$

E-CMP-LST-BASE

$$((\textbf{nil}\ T_1) == (\textbf{nil}\ T_2))\ \ \textbf{n}\ \ \textbf{true} \tag{3.55}$$

E-CMP-LST-NIL1

$$((\textbf{nil}\ T_1) == (\textbf{cons}\ e_{21}\ e_{22}))\ \ \textbf{n}\ \ \textbf{false} \tag{3.56}$$

E-CMP-LST-NIL2

$$((\textbf{cons}\ e_{11}\ e_{12}) == (\textbf{nil}\ T_2))\ \ \textbf{n}\ \ \textbf{false} \tag{3.57}$$

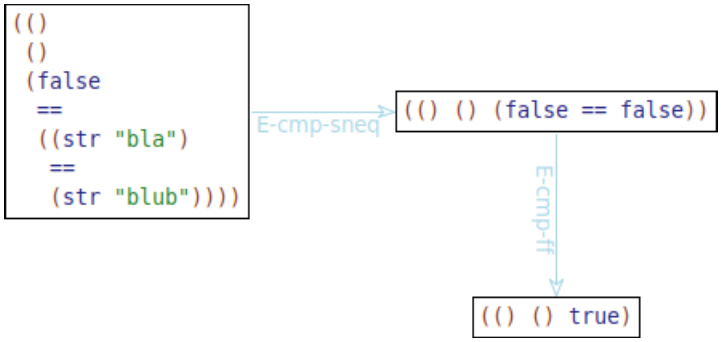


Figure 3.3.: Reduction trace of two nested comparisons. First, we reduce the inner comparison, then the outer one.

E-CMP-LST-CONS
$$((\textbf{cons}\ e_{11}\ e_{12}) == (\textbf{cons}\ e_{21}\ e_{22}))\ \ \textbf{n}\ \ ((e_{11} == e_{21}) \wedge (e_{21} == e_{22})) \tag{3.58}$$

E-CMP-RCD-BASE
$$((\textbf{rcd}\ ()) == (\textbf{rcd}\ ()))\ \ \textbf{n}\ \ \textbf{true} \tag{3.59}$$

E-CMP-RCD-IND
$$\begin{array}{l}((\textbf{rcd}\ ([x_1 = e_1][x_i = e_i]\ldots)) == (\textbf{rcd}\ ([x_j = e_j]\ldots[x_1 = e_2][x_k = e_k]\ldots)))\\ \textbf{n}\ \ ((e_1 == e_2) \wedge ((\textbf{rcd}\ ([x_i = e_i]\ldots)) == (\textbf{rcd}\ ([x_j = e_j]\ldots[x_k = e_k]\ldots))))\end{array} \tag{3.60}$$

For example, the following expression nests two comparisons. The inner comparison compares two string literals. The Boolean value resulting from this comparison is, then, compared to **false**.

$$(\textbf{false} == ((\textbf{str}\ \texttt{"bla"}) == (\textbf{str}\ \texttt{"blub"})))$$

The interpreter evaluates this expression, reducing the inner comparison first. This comparison yields false, since the two string literals are unequal. Next, it reduces the outer comparison comparing false and false. This comparison yields true (see Figure 3.3).

Another kind of Boolean expression are propositional logic operators. Cuneiform provides conjunction, disjunction, and negation. The binary logic operators conjunction and disjunction short-circuit, i.e., if one operand of a conjunction turns out to be false, the whole expression becomes false, no matter if the other operand is a value or not. Similarly, disjunction returns true if any of its operators is true.

E-AND-TT
$$(\textbf{true} \wedge \textbf{true})\ \ \textbf{n}\ \ \textbf{true} \tag{3.61}$$

E-AND-F1
$$(\textbf{false} \wedge e_2)\ \ \textbf{n}\ \ \textbf{false} \tag{3.62}$$

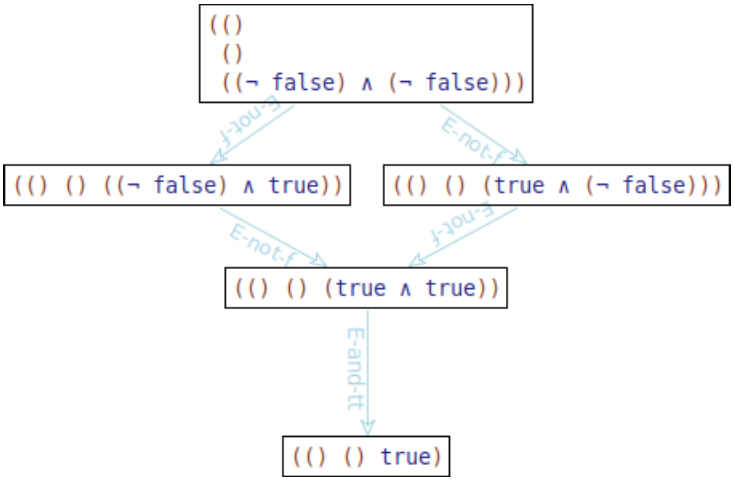


Figure 3.4.: Reduction trace of a Boolean expression made up of conjunction and negation operations. Reduction starts by either reducing the left or the right negation first. Next, we reduce the remaining negation. Last, we reduce the enclosing conjunction.

E-AND-F2

$$(e_1 \wedge \textbf{false}) \;\; \textbf{n} \;\; \textbf{false} \tag{3.63}$$

E-OR-FF

$$(\textbf{false} \vee \textbf{false}) \;\; \textbf{n} \;\; \textbf{false} \tag{3.64}$$

E-OR-T1

$$(\textbf{true} \vee e_2) \;\; \textbf{n} \;\; \textbf{true} \tag{3.65}$$

E-OR-T2

$$(e_1 \vee \textbf{true}) \;\; \textbf{n} \;\; \textbf{true} \tag{3.66}$$

E-NOT-T

$$(\neg\, \textbf{true}) \;\; \textbf{n} \;\; \textbf{false} \tag{3.67}$$

E-NOT-F

$$(\neg\, \textbf{false}) \;\; \textbf{n} \;\; \textbf{true} \tag{3.68}$$

For example, the following expression nests a negation in either operand of a conjunction.

$$((\neg\, \textbf{false}) \wedge (\neg\, \textbf{false}))$$

Here, the interpreter has the freedom to evaluate either negation first. Since short-circuiting a conjunction is possible only if at least one operand is false both negations

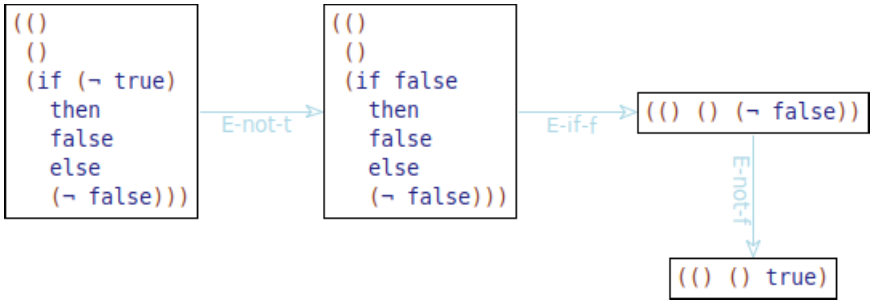


Figure 3.5.: Reduction trace of a conditional. First, we reduce the condition. Next, we replace the conditional expression with the else-branch. Lastly, we reduce the negation that results from the else-branch.

need to be reduced to values until the conjunction operator becomes a reducible expression (see Figure 3.4).

The unary isnil operator returns true if a list is empty and false if it has one or more elements. If the list is non-empty, we do not care whether the list is a value but return false as soon as we can establish that it has the form of a cons. This is similar to short-circuiting Boolean operators.

E-ISNIL-NIL
$$(\textbf{isnil}\ (\textbf{nil}\ T_1))\ \ \mathbf{n}\ \ \textbf{true} \tag{3.69}$$

E-ISNIL-CONS
$$(\textbf{isnil}\ (\textbf{cons}\ e_1\ e_2))\ \ \mathbf{n}\ \ \textbf{false} \tag{3.70}$$

Conditionals

Conditionals allow to branch a workflow depending on whether a condition is true or false. The value of this condition may be known only at runtime, which allows a workflow to adapt the structure of its data dependencies at runtime. A conditional expression has three sub-expressions: The condition, a then-branch, and an else-branch. If the condition evaluates to true, the interpreter replaces the conditional expression with the then-branch. Conversely, if the condition evaluates to false, the else-branch gets selected.

E-IF-T
$$(\textbf{if true then}\ e_1\ \textbf{else}\ e_2)\ \ \mathbf{n}\ \ e_1 \tag{3.71}$$

E-IF-F
$$(\textbf{if false then}\ e_1\ \textbf{else}\ e_2)\ \ \mathbf{n}\ \ e_2 \tag{3.72}$$

For example, the following conditional expression nests a negation in condition position and evaluates to a Boolean.

$$(\textbf{if}\ (\neg\ \textbf{true})\ \textbf{then false else}\ (\neg\ \textbf{false}))$$

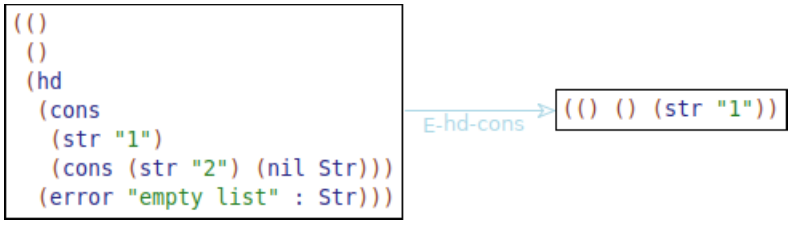


Figure 3.6.: Reduction trace of a hd operator. The interpreter extracts the head of the list operand.

The interpreter evaluates this expression by reducing the condition leaving both branches unevaluated. This reduction produces false and, thus, the else-branch is selected. The interpreter replaces the conditional expression with the negation it finds in the else-branch inside the evaluation context. Last, it reduces the remaining negation (see Figure 3.5).

Lists

Cuneiform provides several operations on lists. First, we consider the list accessors it provides: hd and tl. Both operators have two operands: a list operand and a default operand. The hd operator extracts the head of its list operand and the tl operator extracts the tail of its list operand. If the list operand is empty both operators reduce to their default operand.

$$\text{E-HD-NIL}\quad (\textbf{hd}\ (\textbf{nil}\ T_{12})\ e_2)\ \ \textbf{n}\ \ e_2 \tag{3.73}$$

$$\text{E-HD-CONS}\quad (\textbf{hd}\ (\textbf{cons}\ e_{11}\ e_{12})\ e_2)\ \ \textbf{n}\ \ e_{11} \tag{3.74}$$

$$\text{E-TL-NIL}\quad (\textbf{tl}\ (\textbf{nil}\ T_{12})\ e_2)\ \ \textbf{n}\ \ e_2 \tag{3.75}$$

$$\text{E-TL-CONS}\quad (\textbf{tl}\ (\textbf{cons}\ e_{11}\ e_{12})\ e_2)\ \ \textbf{n}\ \ e_{12} \tag{3.76}$$

For example, the following expression extracts the head of a two-element string list.

$$(\textbf{hd}\ (\textbf{cons}\ (\textbf{str}\ \texttt{"1"})\ (\textbf{cons}\ (\textbf{str}\ \texttt{"2"})\ (\textbf{nil}\ \mathsf{Str})))\ (\textbf{error}\ \texttt{"empty list"}\ :\ \mathsf{Str}))$$

Since the list operand of the hd operator is a literal list and is non-empty, the interpreter reduces this expression by replacing the hd operator with the string literal that constitutes the list head. If the list operand is an expression other than a list literal, e.g., a function application then the interpreter reduces the expression before extracting the head. If the list operand is the empty list the interpreter replaces the hd operator with the default operand.

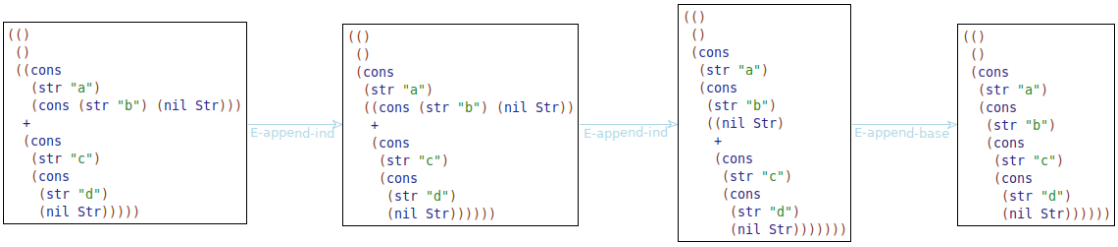


Figure 3.7.: Reduction trace of appending two lists with two elements each. First, we lift out the head of the left list, leaving the tail of the left list as the operand for appending. Next, we lift out the head of the left list again, leaving the empty list as the left list for appending. Last, we replace the append operation with the right list.

We can append lists. Appending two lists requires to inductively apply the two append rules. The base case states that appending the empty list to any expression e_2 yields just e_2. The inductive case states that appending a non-empty list of the form (**cons** e_{11} e_{12}) to any expression e_2 constructs a new list placing e_{11} in its head and appending e_{12} and e_2 in its tail.

$$\text{E-APPEND-BASE}\quad ((\textbf{nil}\ T_1) + e_2)\ \ \textbf{n}\ \ e_2 \tag{3.77}$$

$$\text{E-APPEND-IND}\quad ((\textbf{cons}\ e_{11}\ e_{12}) + e_2)\ \ \textbf{n}\ \ (\textbf{cons}\ e_{11}\ (e_{12} + e_2)) \tag{3.78}$$

For example, the following expression appends two lists holding two elements each.

$$\left(\begin{array}{ll} & (\textbf{cons}\ (\textbf{str}\ \texttt{"a"})\ (\textbf{cons}\ (\textbf{str}\ \texttt{"b"})\ (\textbf{nil}\ \mathsf{Str}))) \\ + & (\textbf{cons}\ (\textbf{str}\ \texttt{"c"})\ (\textbf{cons}\ (\textbf{str}\ \texttt{"d"})\ (\textbf{nil}\ \mathsf{Str}))) \end{array}\right)$$

The interpreter evaluates this expression by inductively lifting out the head of the left list in the append expression until the left operand is the empty list (see Figure 3.7).

For Iteration

One iteration operator is *for*. It allows to iterate over one or more lists by applying some operation element-wise. This can be achieved with a for iteration. Like appending two lists, the for iteration is defined inductively providing a base case and an inductive case. In the base case one or more of the input lists are empty which reduces the for iteration to the empty list.

$$\begin{array}{l}\text{E-FOR-BASE}\\ (\textbf{for}\ T_{\text{body}}\ ([x_i : T_i \leftarrow e_i] \ldots [x_1 : T_1 \leftarrow (\textbf{nil}\ T_2)][x_j : T_j \leftarrow e_j])\ \textbf{do}\ e_{\text{body}}) \\ \qquad\qquad \textbf{n}\ \ (\textbf{nil}\ T_{\text{body}})\end{array} \tag{3.79}$$

In the inductive case the interpreter constructs a new list. The new list's head is the body expression binding the first element of each input list to a variable. The new list's tail is the for iteration with the tails of each of the original input list.

$$\begin{array}{l}\textsc{e-for-ind}\\ \quad (\textbf{for}\ T_{\text{body}}\ ([x_i : T_i \leftarrow (\textbf{cons}\ e_{i1}\ e_{i2})]\ldots)\ \textbf{do}\ e_{\text{body}})\\ \quad \textbf{n}\ \ (\textbf{cons}\ e_{\text{head}}\ (\textbf{for}\ T_{\text{body}}\ ([x_i : T_i \leftarrow e_{i2}]\ldots)\ \textbf{do}\ e_{\text{body}}))\\ \text{when}\ e_{\text{head}} = (\textbf{app}\ (\lambda\ ([x_i : T_i]\ldots)\ (\textbf{ntv}\ e_{\text{body}}))\ ([x_i = e_{i1}]\ldots))\end{array} \tag{3.80}$$

For example, the following for iteration negates each element in a list of three Booleans.

$$\left(\begin{array}{ll}\textbf{for} & \mathsf{Bool}\\ & ([\mathtt{x} : \mathsf{Bool} \leftarrow (\textbf{cons}\ \textbf{true}\ (\textbf{cons}\ \textbf{false}\ (\textbf{cons}\ \textbf{false}\ (\textbf{nil}\ \mathsf{Bool})))))]\\ \textbf{do} & (\neg\ \mathtt{x})\end{array}\right)$$

The interpreter evaluates this expression by constructing a new list in which the head is the iteration body expression binding the free variable `x` to the head element of the input list.

We do not give a reduction trace for this example because it spans twelve reduction steps and has 149 potential reduction states.

Fold Iteration

The second iteration operator is *fold*. Folding over a list allows to update an accumulator for each element in a list. Like append and for iteration we give the semantics of fold iteration as pair of inductive rules with a base case and an inductive case. In the base case, when the iterator list is empty, the fold expression reduces to its accumulator expression.

$$\begin{array}{l}\textsc{e-fold-base}\\ (\textbf{fold}\ [x_{\text{acc}} : T_{\text{acc}} = e_{\text{acc}}]\ [x_1 : T_1 \leftarrow (\textbf{nil}\ T_2)]\ \textbf{do}\ e_{\text{body}})\ \ \textbf{n}\ \ e_{\text{body}}\end{array} \tag{3.81}$$

In the inductive case, when the iterator list is a cons, the fold expression is rewritten: the new accumulator becomes the fold's body expression with the old accumulator and the iterator list's head bound to a pair of variables. The new iterator list becomes the old iterator list's tail.

$$\begin{array}{l}\textsc{e-fold-ind}\\ (\textbf{fold}\ [x_{\text{acc}} : T_{\text{acc}} = e_{\text{acc}}]\ [x_1 : T_1 \leftarrow (\textbf{cons}\ e_{11}\ e_{12})]\ \textbf{do}\ e_{\text{body}})\\ \quad \textbf{n}\ \ (\textbf{fold}\ [x_{\text{acc}} : T_{\text{acc}} = e'_{\text{acc}}]\ [x_1 : T_1 \leftarrow e_{12}]\ \textbf{do}\ e_{\text{body}})\\ \text{when}\ e'_{\text{acc}} = (\textbf{app}\ (\lambda\ ([x_{\text{acc}} : T_{\text{acc}}][x_1 : T_1])\ (\textbf{ntv}\ e_{\text{body}}))\ ([x_{\text{acc}} = e_{\text{acc}}][x_1 = e_{11}]))\end{array} \tag{3.82}$$

For example, the following fold iteration reverts a list of two string elements.

$$\left(\begin{array}{ll}\textbf{fold} & [\mathtt{acc} : (\mathsf{Lst}\ \mathsf{Str}) = (\textbf{nil}\ \mathsf{Str})]\\ & [\mathtt{x} : \mathsf{Str} \leftarrow (\textbf{cons}\ (\textbf{str}\ \texttt{"a"})\ (\textbf{cons}\ (\textbf{str}\ \texttt{"b"})\ (\textbf{nil}\ \mathsf{Str})))]\\ \textbf{do} & (\textbf{cons}\ \mathtt{x}\ \mathtt{acc})\end{array}\right)$$

Figure 3.8.: Reduction trace of a fold iteration reversing a list of two elements. Evaluation takes nine steps.

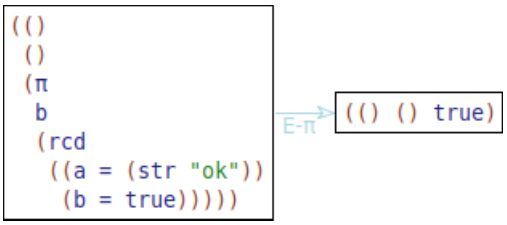


Figure 3.9.: Reduction trace of a record projection. We replace the projection expression with the selected field.

The interpreter evaluates this expression by evaluating the input list and binding the iterator variable `x` to the head of the input list. It uses the resulting expression as the accumulator expression in the next iteration step. Figure 3.8 gives an overview over the reachable reduction states.

Projection

Projection allows us to access a record field. Herein, reduction replaces the projection expression with the expression associated with the selected label.

$$\begin{array}{l}\text{E-}\pi \\ (\pi\ x_1\ (\textbf{rcd}\ ([x_i = e_i] \ldots [x_1 = e_1][x_j = e_j] \ldots)))\ \ \textbf{n}\ \ e_1\end{array} \tag{3.83}$$

For example, the following example projects out the field labeled `b` from a record.

$$(\pi\ \texttt{b}\ (\textbf{rcd}\ ([\texttt{a} = (\textbf{str}\ \texttt{"ok"})][\texttt{b} = \textbf{true}])))$$

The interpreter evaluates this expression by evaluating the operand until it is a record constructor, which is already the case here. Then it replaces the projection expression with the selected field inside the evaluation context (see Figure 3.9).

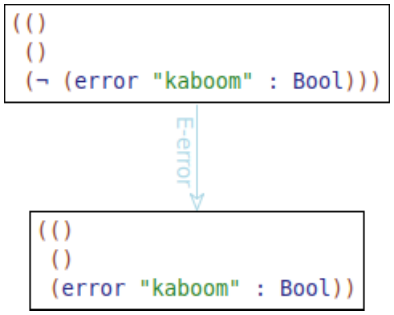


Figure 3.10.: Reduction trace of an error expression in a Boolean negation. We dispose of the evaluation context, replacing the control string with the error expression.

3.5.2. Reduction Relation

Reduction in an Evaluation Context

The notion of reduction we introduced in Section 3.5.1 treats only reducible expressions. In general, however, expressions are nested and the notion of reduction does not explain how to traverse nested expressions to find a reducible expression. Also, the notion of reduction is defined on expressions but we want to reduce programs.

Thus, the following rule states that if the notion of reduction reduces an expression e_1 to e_2 and e_1 appears inside the evaluation context E then e_1 can be replaced by e_2 inside that evaluation context. We look for evaluation contexts and reducible expressions in the control string position of a program.

E-NOTION

$$((e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots)\ E[e_1]) \longrightarrow ((e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots)\ E[e_2]) \qquad \text{when } e_1\ \mathbf{n}\ e_2 \tag{3.84}$$

Errors

When an error appears in an evaluation context the reduction relation disposes of all foreign function applications in the out-box and promotes the error to be the control string. Thereby it drops the evaluation context. An error expression is the only kind of expression that terminates evaluation but is not a value.

E-ERROR

$$((e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots)\ E[(\mathbf{error}\ s_1 : T_1)]) \longrightarrow (()\ ([e_{j1}\ e_{j2}] \ldots)\ (\mathbf{error}\ s_1 : T_1)) \tag{3.85}$$

For example, the following program contains an error expression nested in a negation:

$$(\neg\ (\mathbf{error}\ \texttt{"kaboom"} : \mathsf{Bool}))$$

Here, we apply the rule E-ERROR in the evaluation context $(\neg\ [])$ dropping the negation. Since the out-box is already empty it stays unchanged (see Figure 3.10). Unloading the program yields the plain error expression (**error** `"kaboom"` : Bool).

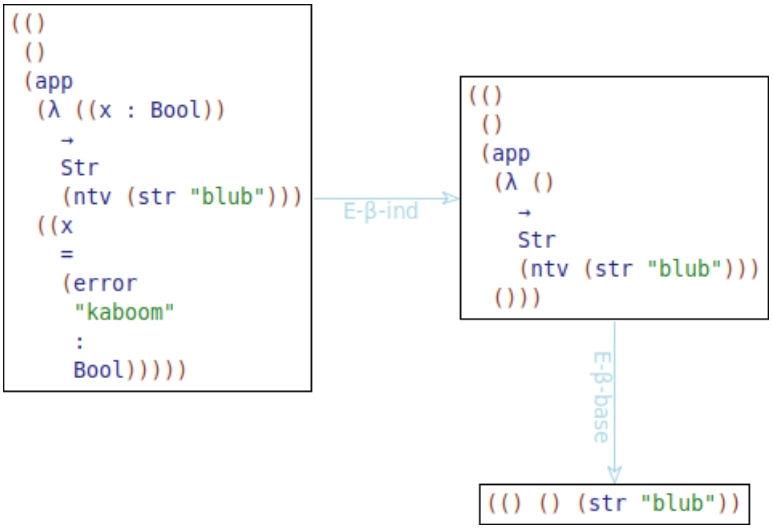


Figure 3.11.: Reduction trace of an error expression in argument position of a native function application. We substitute the error expression into the function body. Evaluating the function yields a string value. The error never appears in a valid evaluation context.

An error expression that appears nested in the control string of a program not always results in an error. Consider the following program:

$$\left(\begin{array}{ll} \textbf{app} & (\lambda\ ([\texttt{x} : \mathsf{Bool}])\ (\textbf{ntv}\ (\textbf{str}\ \texttt{"blub"}))) \\ & ([\texttt{x} = (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool})]) \end{array}\right)$$

Here, a native function application has an error expression in an argument position. The reduction relation evaluates native function applications Call-by-Name. I.e., it replaces each occurrence of the bound variable `x` inside the function body (**str** `"blub"`) with the argument (**error** `"kaboom"` : Bool) without reducing the argument first. Since `x` does not occur in the function body the error vanishes and the application reduces to the string literal (**str** `"blub"`) (see Figure 3.11).

The reduction relation drops an error expression when, in any reachable state, there is no way to construct an evaluation context around the error. In general, it can drop an error if the error appears (i) in a function body and the function is never used, (ii) in argument position of a native function application and the bound variable is never used, (iii) in the then- or else-block of a conditional and evaluation picks the other block, (iv) in the body of a for-iteration that iterates over an empty list, or (v) in the body of a fold-iteration that iterates over an empty list. These cases follow directly from the way an evaluation context can be constructed (see Section 3.4.2).

Errors introduce another caveat. Because of the non-determinism in the evaluation order there are situations where an error might or might not halt evaluation. Consider the following program:

$$(\textbf{true} \vee (\textbf{error}\ \texttt{"kaboom"} : \mathsf{Bool}))$$

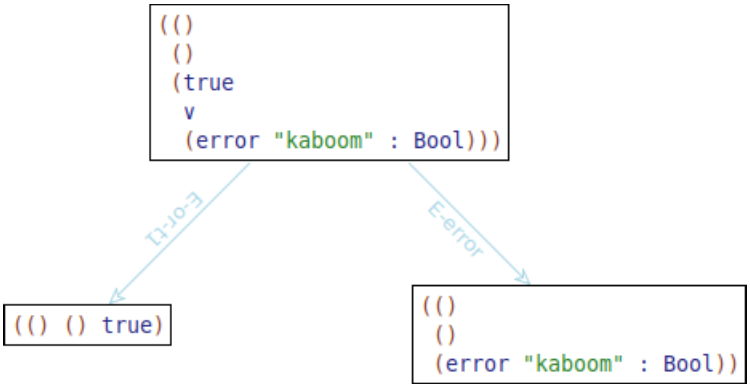


Figure 3.12.: Reduction trace of an error expression in a disjunction operand. An evaluation context can be constructed in two ways leading to two different results.

Here, there are two ways to reduce the control string: (i) the whole control string is a reducible expression in the evaluation context []. We can use the rule N-OR-T1 and get **true** as a result. (ii) The right operand also is a reducible expression in the evaluation context (**true** $\vee$ []). We can use the rule E-ERROR and get the error expression as a result (see Figure 3.12).

This example shows that the Church-Rosser theorem [43] does not hold for Cuneiform as defined here. Note that we can get the Church-Rosser property to hold by imposing an order on evaluating the operands of disjunctions and other operations. This would, however, decrease the potential for parallelism.

Sending

The interpreter requests to execute a foreign function application e_1 by adding the foreign function application to the out-box of a program and by marking the foreign function application with a future.

$$\begin{array}{l}\text{E-}\sigma \\ ((e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots)\ E[e_1]) \longrightarrow ((e_1\ e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots)\ E[(\textbf{fut}\ e_1)]) \\ \quad \text{when } e_1 = (\textbf{app}\ (\lambda\ ([x_k : T_k] \ldots)\ (\textbf{frn}\ x\ T_{\text{ret}}\ l\ s))\ ([x_m = v_m] \ldots))\end{array} \tag{3.86}$$

Receiving

The execution environment replies with the result of foreign function applications by adding an expression-value pair to the in-box of a program. Herein, the expression is a foreign function application and the value is the datum that results from executing the application.

The following rule states that whenever a future marking a foreign function application (**fut** e_1) appears in an evaluation context in the control string of a program and its result

v_1 is known then the future can be replaced with the result.

$$\begin{array}{l}\text{E-}\rho \\ ((e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots [e_{11}\ e_{12}][e_{k1}\ e_{k2}] \ldots)\ E[(\textbf{fut}\ e_1)]) \\ \longrightarrow ((e_i \ldots)\ ([e_{j1}\ e_{j2}] \ldots [e_{11}\ e_{12}][e_{k1}\ e_{k2}] \ldots)\ E[v_1])\end{array} \tag{3.87}$$

3.6. Derived Forms

A derived form, or syntactic sugar, is a form that we define using other forms instead of extending the language's semantics. To define the meaning of derived forms we introduce the binary expansion relation `x`. The relation `x` defines how macro expansion works in Cuneiform. In the implementation the expansion relation is part of the parser and it applies the expansion rules wherever possible. In most general purpose programming languages, macro expansion is a processing step in its own right. In Cuneiform we integrated it into the parser because there are only two straight-forward derived forms.

3.6.1. Let Binding with Pattern Matching

When we compose large workflows out of smaller parts, we often need to bind an expression to a name. So, in the rest of the workflow script, we can use the name instead of having to write the whole expression. This is useful not just when we want to reuse expressions but also to organize our thoughts by writing one small expression at a time instead of one large expression.

Many SQL dialects provide the `with` clause to define named sub-queries. In programming languages we often find let bindings for that purpose. Some functional programming languages like ML, Racket, or Erlang also provide let bindings with pattern matching. Pattern matching allows us to access composite data structures by restating how they are constructed instead of deconstructing them with accessors. Pattern matching also allows us to bind several names at once.

For this reason, Cuneiform provides let bindings with pattern matching. Herein, a pattern can be either a variable name, binding the expression as is, or a record whose fields contain patterns. This way, it is easy to access nested records or to access multiple record fields in one go.

To define let bindings with pattern matching we first need to define what form a pattern can have:

$$\begin{array}{rcl} r & ::= & x : T \\ & & (\textbf{rcd}\ ([x = r] \ldots)) \end{array} \tag{3.88}$$

Next, we extend the definition of expressions to also entail let bindings with patterns as their left-hand-side. In the let form, the first expression is the expression to bind and the second expression is the expression in which the binding can be used.

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & \textbf{let}\ r = e\ ;\ e \end{array} \tag{3.89}$$

Next, we state how to evaluate a let binding. We do this by extending the notion of reduction with the following three rules. The first rule states that binding a name x_1 of type T_1 to an expression e_1 in the body expression e_2 is the same as applying a function with one argument x_1 where e_1 is the argument and e_2 is the function's body:

$$\begin{array}{rl} & \textbf{let } x_1 : T_1 = e_1 \ ; \ e_2 \\ \text{x} & (\textbf{app } (\lambda \ ([x_1 : T_1]) \ (\textbf{ntv } e_2)) \ ([x_1 = e_1])) \end{array} \tag{3.90}$$

Next, we give the base case for record patterns. Binding an empty record pattern evaluates to just the body expression e_2, dropping the binding expression e_1:

$$\begin{array}{rl} & \textbf{let } (\textbf{rcd } ()) = e_1 \ ; \ e_2 \\ \text{x} & e_2 \end{array} \tag{3.91}$$

Last, we give the inductive case for record patterns. It states that when we have a non-empty record, we create a new let binding that projects out the field x_0 of the binding expression e_1. The body expression of the new let binding is the original let binding but reduced by the record field we just projected out.

$$\begin{array}{rl} & \textbf{let } (\textbf{rcd } ([x_0 = r_0][x_i = r_i] \ldots)) = e_1 \ ; \ e_2 \\ \text{x} & \textbf{let } r_0 = (\pi \ x_0 \ e_1) \ ; \ \textbf{let } (\textbf{rcd } ([x_i = r_i] \ldots)) = e_1 \ ; \ e_2 \end{array} \tag{3.92}$$

3.6.2. Recursive Function Definition

Cuneiform's abstract language definition introduces a fixpoint operator to enable recursive functions. However, a user expects that a function can be used inside this function's body without the need of additional annotation. Accordingly, we introduce the `letrec` form that takes allows to define a recursive function and also binds its name in the subsequent expression. First, we extend the expression syntax with the `letrec` form:

$$\begin{array}{rcl} e ::= & \ldots & \\ & | & \textbf{letrec } x \ ([x : T] \ldots) \rightarrow T \ \{e\} \ e \end{array} \tag{3.93}$$

Next, we define how the `letrec` form expands to a `let` form that binds a fixpointed function.

$$\begin{array}{rl} & \textbf{letrec } x_f \ ([x_i : T_i] \ldots) \rightarrow T_{\text{ret}} \ \{e_{\text{body}}\} \ e_2 \\ \text{x} & \left(\begin{array}{l} \quad \textbf{let} \quad x_f : (\mathsf{Fn} \ ([x_i : T_i] \ldots) \rightarrow T_{\text{ret}}) = \\ \qquad\quad (\textbf{fix } (\lambda \ ([x_f : (\mathsf{Fn} \ ([x_i : T_i] \ldots) \rightarrow T_{\text{ret}})][x_i : T_i] \ldots) \ (\textbf{ntv } e_{\text{body}}))); \\ e_2 \end{array} \right) \end{array} \tag{3.94}$$

3.7. Further Reading

A formal specification of syntax or semantics is lacking for the majority of workflow languages in use today. However, for a number of languages formal models have been

developed, notably Pegasus [36] and Kepler [85, 160, 252]. A scientific workflow language with an extensive body of work regarding its semantics is Taverna [117]. Taverna is a graphical scientific workflow language targeting users in bioinformatics and other life sciences. It focuses on the integration of heterogeneous software and web-services. The original formulation of Taverna's semantics [226] is formulated as a natural semantics based on computational lambda calculus [163]. It was succeeded by a number of refinements formulated as state transitions with trace semantics [111, 209, 210, 211] pushing the understanding of Taverna's semantics in a process-oriented direction. The publications characterizing Taverna's semantics emphasize the fact that Taverna services can have side effects or be non-deterministic, i.e., that the order in which services are invoked potentially influences the workflow result.

The idea that data dependencies in scientific workflows can be expressed in lambda calculus has been formulated several times. Ludäscher and Altintas presented a way to express scientific workflows in Haskell syntax [151] and observed that parallelization is directly derivable. Kelly et al. [129, 127] have defined data dependencies among web-services directly in untyped lambda calculus. Cuneiform differs from Kelly's approach in that we make a minor modification to the canonical presentation of the simply typed lambda calculus allowing the uniform notation of abstractions (native functions) and external operators (foreign functions).

Apart from this modification, we stick as closely as possible to a simply typed lambda calculus. However, existing scientific workflow languages are rooted in various formalisms. E.g., Pig Latin is inspired by SQL and is, thus, rooted in relational algebra. Taverna started out with a functional formulation but turned in the direction of trace semantics. Kepler emphasizes its relationship with process networks and the actor model but its orchestrator concept makes execution behavior actually exchangeable. Nextflow [56] has been designed around the concept of channels and is, therefore, closely related to process calculi like CSP. Finally, some workflow languages have been designed around Petri Nets, e.g., Grid-Flow [94].

Similar to Turi et al. we introduce Cuneiform's semantics by first introducing its syntax and typing rules and then discuss its evaluation rules. In contrast, we introduced those evaluation rules in the form of a *structural operational semantics* [187] which defines evaluation as the repeated application of small-step evaluation rules. An alternative way to present an operational semantics is the form of *natural semantics*, which defines evaluation of a program in a single big step [126]. Yet another candidate would have been a denotational semantics which transforms an expression until it is composed only of symbols and operations with an intuitive interpretation [213].

Cuneiform's approach and parallel execution is is inspired by distributed functional programming languages like Eden [34, 149] or the distributed Haskell implementation GDH [188]. Its take on large-scale data analysis on top of a distributed file system is inspired by MapReduce [50, 240] and Spark [248, 247]. Eventually, Cuneiform's integration of external software is inspired by scientific workflow languages like Taverna [117] or Galaxy [87]. However, a language combining these advantages in a large-scale functional language that is agnostic about a function body's implementation language has, to our knowledge, not been otherwise conceived.

4. Distributed Execution Environment

In Chapter 3 we introduce Cuneiform's abstract syntax and semantics. We have give Cuneiform's semantics in a way such that it involves communication with an execution environment. The Cuneiform interpreter that enacts the semantics sends a foreign function application to the execution environment and receives the result of a foreign function application when the execution environment has executed the application. Herein, the Cuneiform interpreter does not wait for a single application to finish but attempts to discover more applications in the workflow expression to send to the execution environment. Upon receiving the result of an application the interpreter has access to new knowledge unlocking more applications to send. Furthermore, we assume that each application that the execution environment receives is deterministic and independent from any other application. This means that caching application results and executing applications in parallel cannot influence the overall result of the workflow.

In this Chapter, we introduce the components that make up a distributed execution environment suitable to handle interact with a Cuneiform interpreter. Figure 4.1 shows an overview of these components. We give each component's behavior as a Petri net. To motivate the communication patterns among the execution environment's components we give a simple example workflow. This workflow exposes parallelism potential. Next, we discuss the communication patterns exposed by this concrete scenario in terms of a UML sequence diagram. We rewrite this sequence diagram to a distributed run which allows us to outline the components involved in creating such a run in terms of their interfaces. We take this outline as a scaffold and fill in the missing parts one component at a time.

4.1. Distributed Scenario

Before we discuss the distributed execution environment in detail, we give an example workflow to see how messages flow between the execution environment's components. The following Cuneiform script defines a function `wc`, which counts the words in a given text file. It takes one argument `file` and produces a record with one field `n`. The word count is given as a string. Next, we define a list literal `filelst` containing four file literals. Lastly, we iterate over the list calling the `wc` function for each file. The result is a list with four string elements, each stating the number of words found in the corresponding file.

```
def wc( file : File ) -> <n : String> in Bash *{
  n=‘wc -w $file | awk ’{print $1}’‘
}*
```

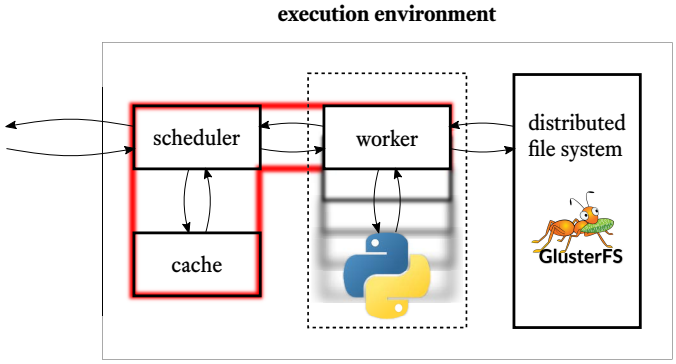


Figure 4.1.: Distributed execution environment components. The scheduler exchanges messages with a Cuneiform interpreter. The execution environment consists of a scheduler, a cache, and several worker instances. The worker instances, in turn, access a distributed file system and drive other scripting languages, e.g. Python.

```
let filelst : [File] =
  ['a', 'b', 'c', 'd' : File];

for f : File <- filelst do
  wc( file = f ) : String
end
```

Translating the above Cuneiform script into its abstract syntax yields the following expression that conveys the entire workflow:

$e_0 ::=$
let `wc` : (Fn ([`file` : File]) → (Rcd ([`n` : Str]))) =
(λ ([`file` : File]) (**frn** `wc` (Rcd ([`n` : Str])) Bash `"n=‘wc -w $file | awk ..."`));
let `filelst` : (Lst File) =
(**cons** (**file** `"a"`) (**cons** (**file** `"b"`) (**cons** (**file** `"c"`) (**cons** (**file** `"d"`) (**nil** File)))));
(**for** Str ([`f` : File ← `filelst`]) **do** (**app** `wc` ([`file` = `f`])))

Internally, the Cuneiform interpreter loads this expression into the control string position of a program with empty out- and in-box. The initial program to be interpreted looks like so:

$$p_0 ::= (() \ () \ e_0)$$

Applying the reduction relation to the program p_0 until it cannot be further reduced produces the program p_1 which contains each of the four applications of the `wc` function in its outbox:

$$p_1 ::= ((e_{11} \ e_{12} \ e_{13} \ e_{14}) \ () \ e_1)$$

When all four foreign function applications have been sent away, scheduled, executed, and returned we end up with a program with an empty out-box and the results of all

four applications in its in-box.

$$p_1' ::= (() \ ([e_{11}\ e_{21}]\ [e_{12}\ e_{22}]\ [e_{13}\ e_{23}]\ [e_{14}\ e_{24}])\ e_1)$$

Lastly, substituting the result for each of the foreign function applications in the appropriate place in e_1 gives us e_2 which has the form of a literal list of strings.

$$p_2 ::= (() \ ([e_{11}\ e_{21}]\ [e_{12}\ e_{22}]\ [e_{13}\ e_{23}]\ [e_{14}\ e_{24}])\ e_2)$$

Since e_2 is also a value, the interpreter stops computation, discards its program state and returns e_2 to the user.

From the viewpoint of the distributed execution environment the transition from p_1 to p_1 is interesting because in this step the application requests in the out-box become application-value pairs in the in-box of the interpreter. This transition is outside of the control of the interpreter.

The separation of the interpreter and the execution environment is important because the interpreter is defined in terms of an operational semantics. But operational semantics make progress only in a stepwise manner where each step depends on the outcome of the previous step. Thus we can construct a system that is capable of distributed, independent operations only from a specification that allows us to compose a system of independent components. The in-box and the out-box of the interpreter's program state represent the interface between the language interpreter that we define as an operational semantics and the distributed execution environment that we define as a Petri net.

However, before we discuss the Petri net itself, let us see how the system passes messages among components. Figure 4.2 shows a sequence diagram where the user passes the workflow e_0 to the interpreter c_1. As outlined above from the perspective of the interpreter, the workflow expression e_0 is loaded into the program p_0 and becomes evaluated to p_1. In p_1 there are four foreign function applications ready to be sent in the out-box of the program. The sequence diagram shows how each application is sent to the scheduler m which then decides which worker w_1 or w_2 gets to execute which foreign function application. When all applications are executed and their results have been communicated back to the interpreter c_1 the program p_1' can be evaluated to p_2 which is the final state of the interpreter. As a last step, the interpreter discards its state and returns e_2 to the user which is a value. Note that the programs p_i are internal to the interpreter. What the interpreter sends and receives from and to other components are either workflows (which are expressions) or foreign function applications (which are also expressions).

As a next step, we take the UML sequence diagram shown in Figure 4.2 and rewrite it as a distributed run [194]. Figure 4.3 shows the result of this rewriting. We have tried to conserve the geometric structure of the sequence diagram having events flow generally from the top to the bottom and having messages sent into the system going right and messages that return from the system going left. Strictly, all places need to have an inscription on them. However, we used inscriptions only for the interface places where they represent the messages passed among components. The places that are left without inscription represent the internal state of either the interpreter (left column) or

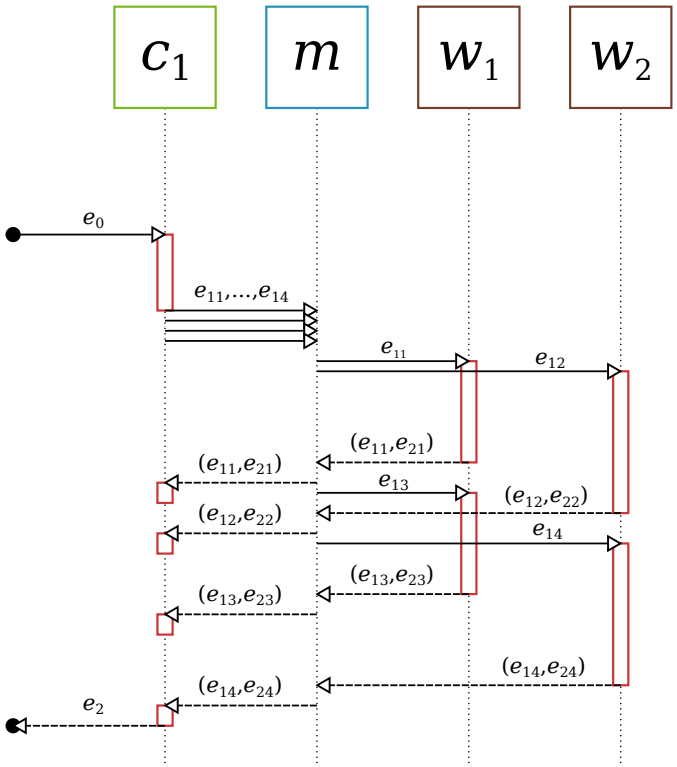


Figure 4.2.: Sequence diagram showing a workflow execution scenario completing four independent applications

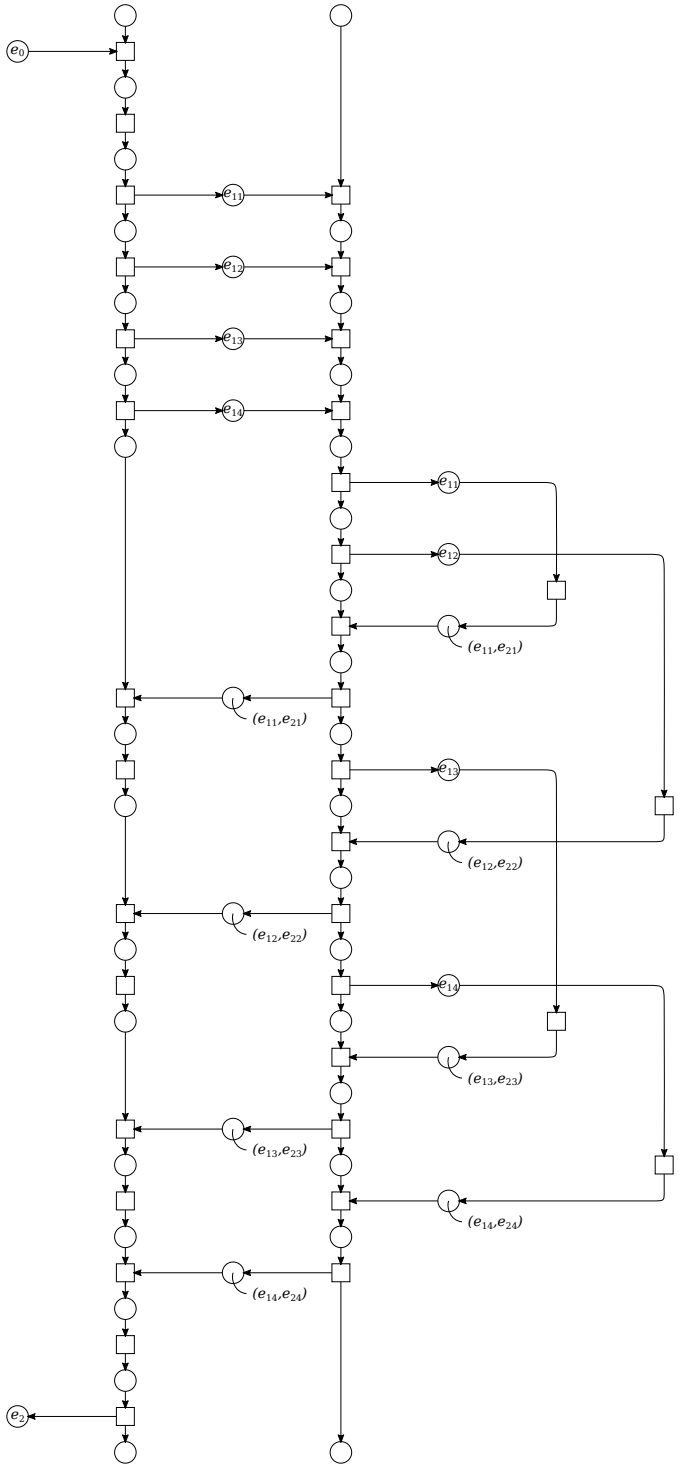


Figure 4.3.: Distributed run corresponding to the previous sequence diagram. Actions, represented by squares, on the same vertical line occur in the same service.

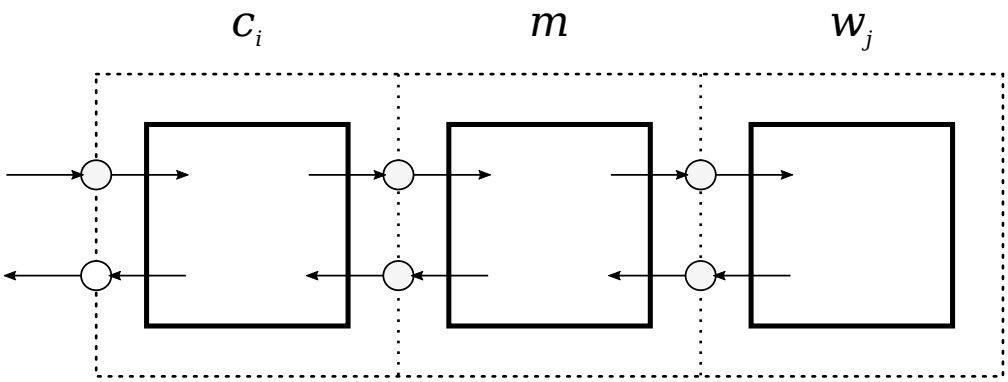


Figure 4.4.: Outline of the distributed execution environment. From left to right the interpreter c_i, the scheduler m and the worker w_j.

the scheduler (middle column). Workers are shown here as stateless although we refine them to stateful processes later in this chapter.

The distributed run shown in Figure 4.3 can be interpreted as the unfolding of a place-transition net. Merely finding a folding of this distributed run would cut it too short because the run illustrates only the system's communication structure on a high level without explaining important aspects like caching, fault recovery, or scheduling. Rather, what we are looking for is an idea of what nets that expose this communication structure have in common. What we can learn from the distributed run is that 1. an interpreter can receive a workflow expression from the user, 2. the interpreter can send a foreign function application to the scheduler, 3. the scheduler can send an application to a worker, 4. a worker can send a result to the scheduler, 5. the scheduler can send a result to the interpreter, and 6. the interpreter can send a value expression to the user. In Figure 4.4 we show an outline of a open place-transition schema net that fulfills just the above six properties. Herein, components communicate providing a place for each of the above six communication activities.

In the remainder of this chapter we describe what exactly the inner workings of each of the three component types: interpreter, scheduler, and worker. However, before we go into detail we give the net in full in Figure 4.5.

In the following sections we delineate each component of the distributed execution environment that runs Cuneiform as a place-transition schema net. We begin with the interpreter in Section 4.2, followed by the scheduler in Section 4.3, and ending with the worker in Section 4.4.

4.2. Interpreter

An interpreter hosts Cuneiform's semantic model. It communicates with both the user and the scheduler component (see Section 4.3). On the user side, the interpreter receives a workflow expression and returns a value or an error. Upon receiving a workflow expression from the user the interpreter first loads the expression into a program. It then

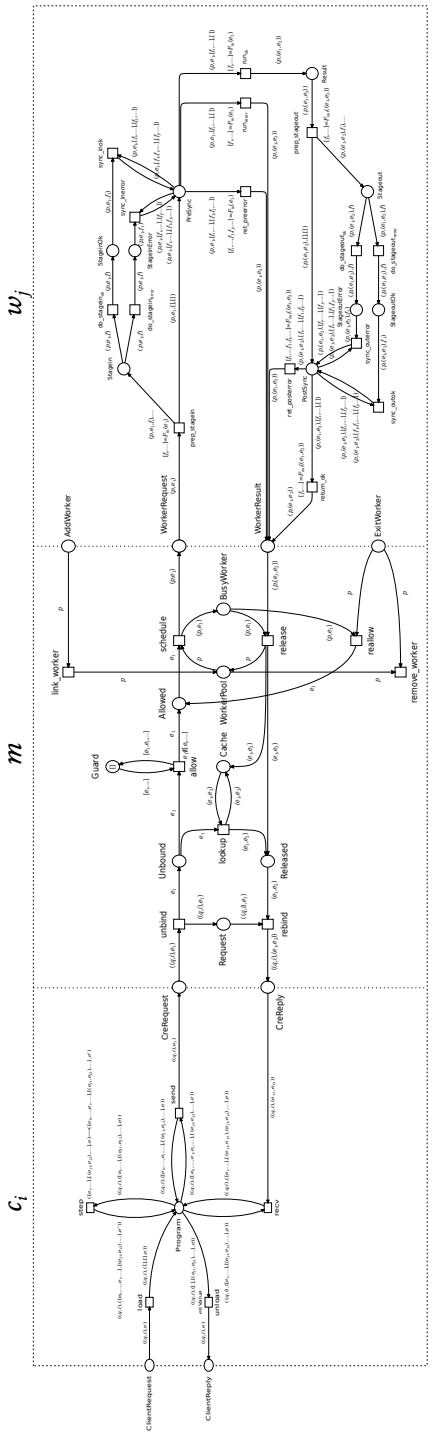


Figure 4.5.: Petri net composing three service types from left to right: interpreter, scheduler, and worker. A detail of each service is provided in the following figures.

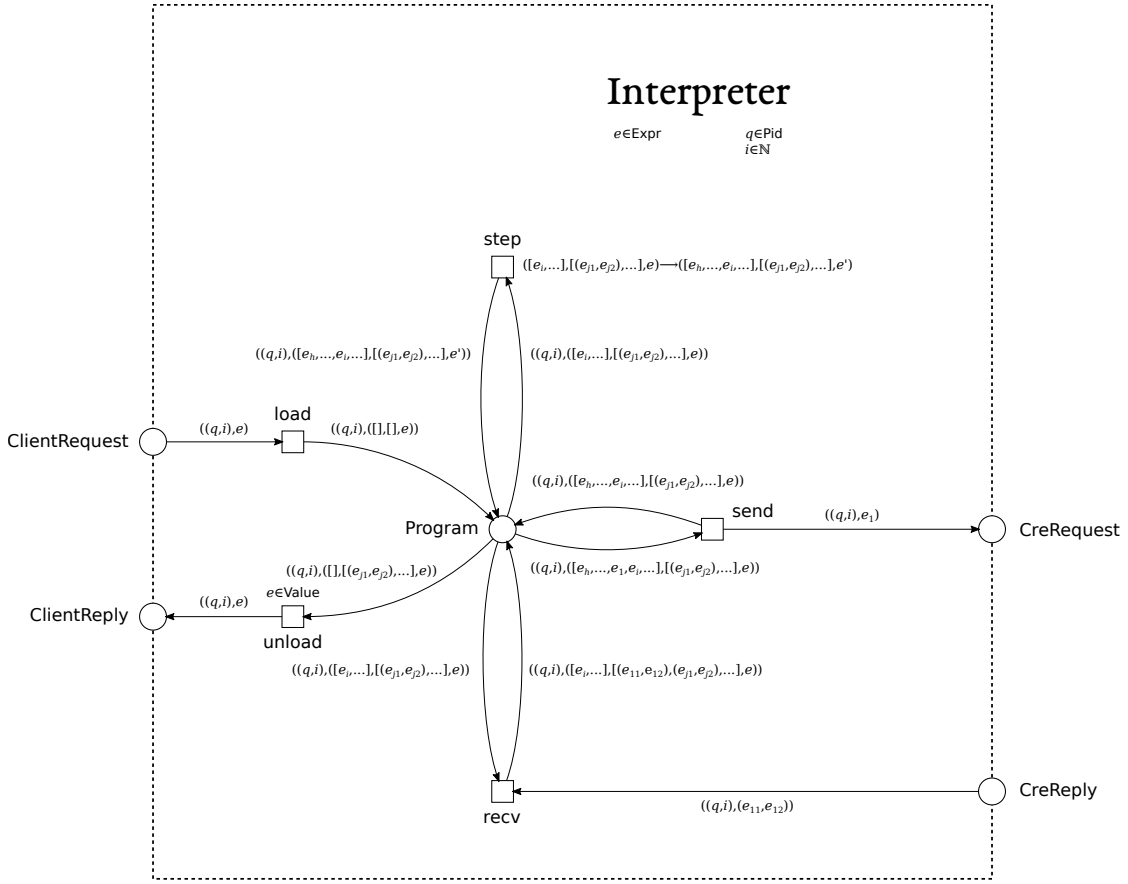


Figure 4.6.: Petri net model of the interpreter component. The user interacts with the interpreter on its left interface. The right interface allows the interpreter to exchange messages with the scheduler.

evaluates the program until it no further evaluation steps are possible (see Chapter 3). In this event, the interpreter returns the final result of evaluation.

Many reduction rules alter only the control string position of the program. Only the rules E-σ for sending and the E-ρ for receiving interact with other components of the distributed execution environment. Thus, the interpreter has two transitions, send and recv that mirror the aforementioned reduction rules, thus, embedding the Cuneiform semantics in the interpreter. Herein, the send transition removes a foreign function application expression from the out-box of the program and sends it to the scheduler via the CreRequest place. Upon receiving a result via the CreReply place the client adds the result to the program's in-box by firing the recv transition. Figure 4.6 shows the complete schema net of the interpreter.

4.2.1. Parsing and Consistency Check

In the previous section we take for granted that the workflow expression the user provides is always well-formed and consistent. However, the actual input coming from the user is a general string. In general, this string might not amount to a workflow that actually produces a result. E.g., the user input may contain unrecognized symbols or it may expose unbalanced parantheses. The workflow may use an unbound variable or it may provide a string where only a file can do. We do not show the handling mechanism explicitly in the Petri net model in Figure 4.6. However, the implementation comes to the workflow expression through a processing chain of scanning, parsing, and type checking the user input. Input processing can fail at any stage. For the sake of simplicity we have omitted these steps in the Petri net model of the interpreter.

4.2.2. Detecting Scheduler Failure

We also take for granted that the scheduler service is always available. In a situation where the scheduler fails, the interpreter detects is assumed to detect this failure and terminate immediately. The existence of a scheduler is, moreover, a precondition for the interpreter to start. Both, the initial connection to the scheduler and the detection of its possible failure are implicit in the Petri net model. In the implementation, it is the virtual machine, hosting the interpreter process that makes sure both conditions are met.

4.2.3. Anticipating Foreign Function Application Failure

A foreign function application may fail due to a syntax error in the foreign function body, an exception, or a missing input or output file. In this case the interpreter halts evaluation for the specific workflow the application appears in and makes an error expression the program's control string. The user receives an error message stating the reason the foreign function application failed. The evaluation rules for errors are explained in Chapter 3.

4.2.4. Recovering from Worker Errors

Lastly, a worker instance might fail. The scheduler detects this failure and reschedules any foreign function application a failing worker might have processed (see Section 4.3.3). Rescheduling remains hidden from the user. The only way a worker failure can be detected by the user is through varying execution times for applications of a specific execution duration.

4.2.5. Nondeterminism in Step, Send, and Receive

Note, that the transitions step, send, and recv span a structural decision. I.e., there might be a program on the Program place that enables more than one of these transitions. In this case, it is up to the implementation to decide which transition fires. It is indeed possible to arrive at such a situation. E.g., after uncovering a ready foreign function application, the interpreter may either proceed to evaluation, possibly uncovering another application, or it may send the application to the scheduler first. Similarly, the interpreter may be in the middle of evaluation when an application-result pair arrives on the CreReply place. In this situation the interpreter may either continue evaluation or place the newly arrived result in the in-box of the program first. The non-determinism in the order of making progress on evaluation and sending or receiving foreign function applications has no influence on the workflow result unless the possibility is explicit also in the semantics (see Chapter 3).

4.3. Scheduler

The central hub of the distributed execution environment is the scheduler. To the interpreter it acts as a service that receives a foreign function application and replies with the value, that foreign function application produces upon executing it. Herein, sending the foreign function application and receiving the result are asynchronous, i.e., both the interpreter and the scheduler send messages spontaneously and at their own rate.

The scheduler has a number of worker processes at its disposal. When receiving a new foreign function application, the scheduler picks an idle worker to execute that application. In the case that all workers are busy, the scheduler keeps any incoming foreign function application until a worker becomes idle. In addition, it caches every application and its results. If an interpreter requests an application a second time, the scheduler does not send it to a worker but generates the reply from its cache. This speeds up the execution of similar workflows.

The scheduler can serve several interpreters and use many workers. Thus, the scheduler must react to both failing clients and workers. In particular, an unreachable interpreter must not affect any other interpreter that uses the same scheduler. Also, if a worker fails, all applications associated with it must be rescheduled so that the execution environment appears functional to the interpreter even though workers join and leave the distributed execution environment freely.

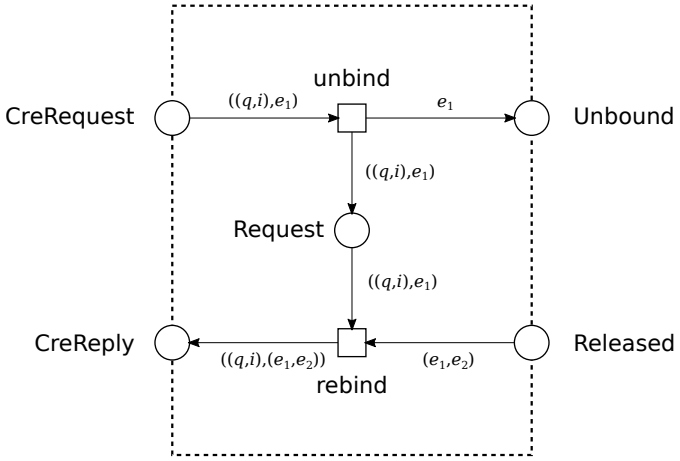


Figure 4.7.: Petri net model of the interpreter communication scheduler sub-component in the scheduler component.

4.3.1. Interpreter Communication

To process a foreign function application, the scheduler receives a pair containing the application itself and an interpreter address to send the reply to. However, other than creating the reply message, the information which interpreter originated an application is inessential. I.e., neither caching, scheduling, nor executing a foreign function application depend in any way on the interpreter address. Thus, upon receiving an application from a interpreter, the first operation the scheduler performs is to separate the application from the originating interpreter's address. Only when the result is ready to be returned to the interpreter, the scheduler re-associates the interpreter address, producing a triple holding the interpreter's address, the original application, and the result of execution.

Figure 4.7 shows the mechanism that unbinds the interpreter address from the application and later rebinds it on the way back to the interpreter. The scheduler receives foreign function applications on the CreRequest place. It stores the pair as is on the Request place and sends only the application to the downstream part of the scheduler via the Unbound place. When execution finishes an application-result pair appears on the Released place. Lastly, the scheduler rebinds the client address to the result-pair and sends the resulting triple to the client via the CreReply place. The net contains no extra circuitry to deal with client failure. Implicitly the scheduler disposes of messages addressed to a failed interpreter.

4.3.2. Cache

Often, in informatics, we can trade memory for speed. E.g., we can cache intermediate results to save time at the expense of memory. It would be ideal to cache only the things that get reused later. But the knowledge of how often something will be reused in the future is hard to come by. However, the utility of caching something that is reused many times often outweighs the cost of also caching things that are used only once.

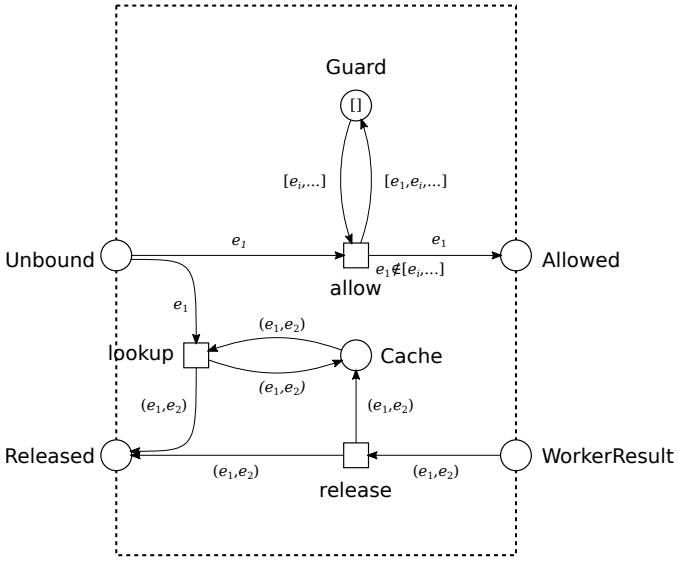


Figure 4.8.: Petri net model of a cache scheduler sub-component in the scheduler component.

In a bioinformatics setting, researchers often explore. They continuously extend, change, or prune a workflow in the process of creating it. This means large portions of a workflow stay constant over editing steps. A workflow system that caches intermediate results shorten turnaround times to just the time needed to compute the difference from the previous workflow version. Also, complex workflows sometimes to accumulate programming anti-patterns like accidental repetitions [44]. If we cache intermediate results we avoid the computational overhead of these repetitions in a workflow. Lastly, if research areas overlap, researchers may profit from sharing cached intermediate results. E.g., in a medical context, indexing the Human reference genome is common. If we cache intermediate results researchers working on the same cluster, applying the same operation to the same data automatically share cached intermediate results.

We can model a cache as a Petri net consisting of two activities: a guard and a look-up. The guard allows a foreign function application to be scheduled only if it is new. The look-up pairs a foreign function application with its result if the result is known. Consequently, a foreign function application coming from a client can face three situations: (i) the application is new, then it is scheduled to compute the result, (ii) the application has been scheduled before and the cached result is available, then the result is drawn from the cache, or (iii) the application has been scheduled before but the result is not yet available, then nothing happens until the result becomes available. Figure 4.8 shows a Petri net of a cache.

Note that for any given marking of the net at most one of the transitions allow or lookup is enabled. A foreign function application can either be fresh or non-fresh, never both. The allow transition is enabled only for fresh applications. Conversely, fresh applications can never appear in the cache, so the lookup transition can be enabled only

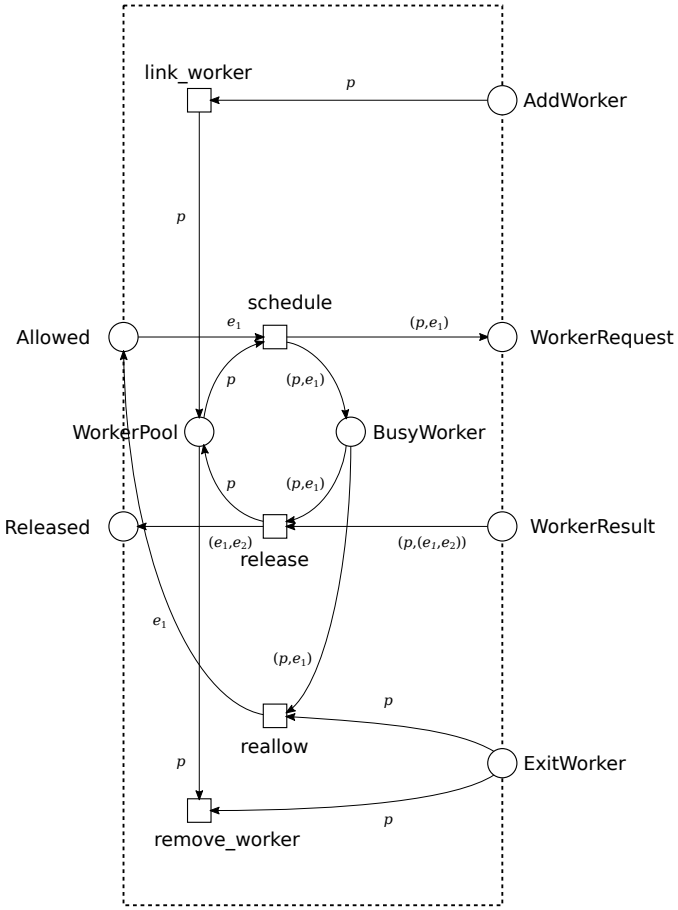


Figure 4.9.: Petri net model of the worker communication scheduler sub-component in the scheduler component.

for non-fresh applications.

While it is important to note that scheduling and cache look-up are mutually exclusive ways to serve an application, we also want to know that any application is served at some point. By close observation we notice that after execution of a fresh application has finished, the application-result pair appears in the cache. From this point on, the scheduler serves all non-fresh duplicates of that application from the cache. Thus, assuming that at least one worker is available and that every foreign function application terminates, the lookup transition is enabled at some point for any non-fresh application.

4.3.3. Scheduling

Distributing independent foreign function applications requires us to manage the life cycle of both the application that need to be scheduled to a worker and the life cycle of the workers that join the worker pool as part of their initialization and may leave the worker pool ensuring that a leaving worker never affects the health of the system

including the workflows running on it.

Figure 4.9 shows a Petri net model of a scheduling subcomponent. The scheduling subcomponent receives a new foreign function application e_1 on the Allowed place and sends its associated result via the Release place. To schedule an application, the scheduling subcomponent associates the application with a worker address p and notifies that worker by creating a message token on the WorkerRequest place. Workers that have finished processing an application send a triple containing the worker address p, the original application e_1, and the result expression e_2. This triple appears on the WorkerResult place. Prior to returning the application result to the upstream subcomponents of the scheduler the scheduling subcomponent unbinds the worker address from the result in the release transition. Thereby, the worker address returns to the WorkerPool place marking it as idle. A pair containing only the original application and the result value appears on the Release place from where it moves upstream.

4.3.4. Recovering from Worker Disconnection

While the interpreter and distributed execution environment process a workflow, workers can join and leave. However, the result of a workflow must be deterministic, no matter how many workers disconnect or what application a worker processes when it disconnects. Thus, we need to register a worker when it starts up and to detect when it disconnects. Furthermore, when a worker that processes an application disconnects, the scheduler needs to reschedule that application.

When the scheduler component starts, it knows about neither interpreters nor workers. Conversely, an interpreter can start only if it knows which scheduler to address for foreign function applications and also a worker can start only if it knows which scheduler to expect applications from. When a worker starts, it registers with the scheduler by making its address appear on the AddWorker place. The worker address can then be added to the WorkerPool via the link_worker transition.

When a worker becomes unreachable, its address appears on the ExitWorker place. If the worker is currently idle, the remove_worker transition removes it from the WorkerPool. If, however, the worker is currently busy, then the reallow transition removes it from the BusyWorker place and puts the foreign function application back on the Allowed place so that it can be executed by a different worker.

Note that for any given worker either remove_worker or reallow is enabled but never both. The reason is that a worker cannot be idle and busy at the same time.

There is an edge case, where the scheduler processes requests from an interpreter but has no workers registered to it. This situation can arise at scheduler initialization, when no worker had the chance to register yet or if all workers have failed. In such a situation the workflow cannot make progress. However, the system is still in a consistent state because in the future, a worker might register, which would allow the distributed execution environment to make progress again.

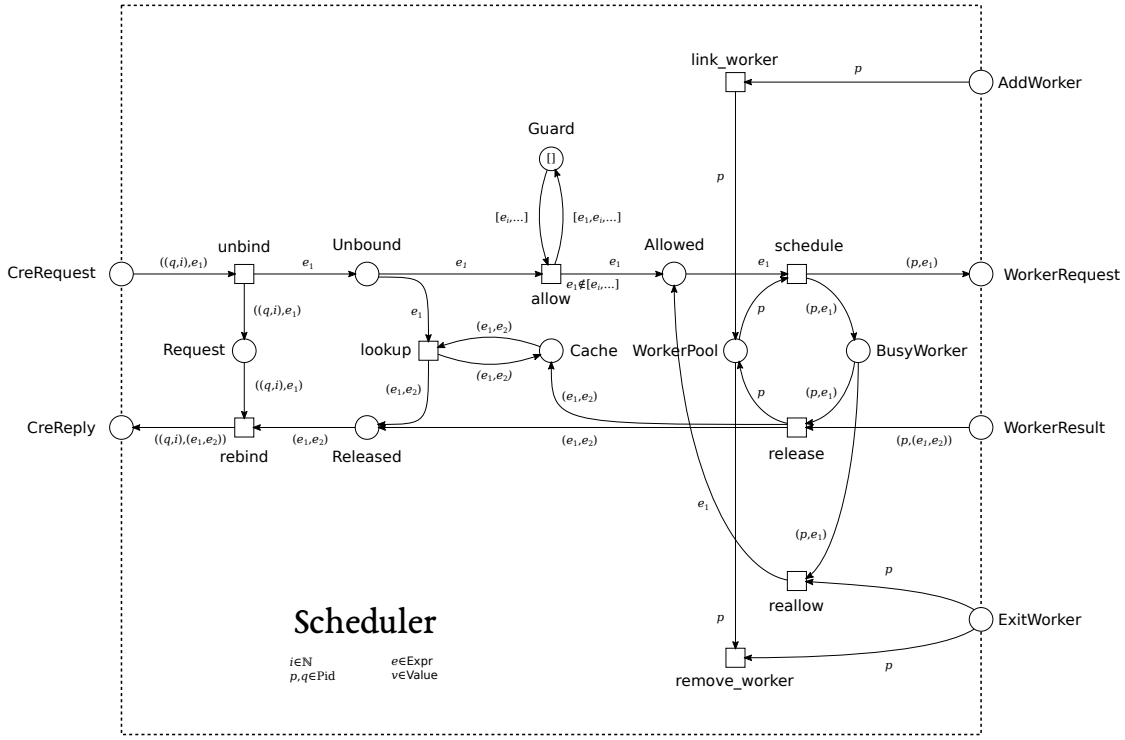


Figure 4.10.: Petri net model of the scheduler component. The model composes subcomponents for interpreter unbinding, caching, and scheduling.

4.3.5. Scheduler Composition

As a last step, we compose the previously discussed subcomponents, the interpreter communication subcomponent, the cache subcomponent, and the scheduling subcomponent, to form the scheduler. Figure 4.10 shows the result of this composition. It is straightforward to compose these subcomponents since each follows the same general layout: On the top path going left to right each subcomponent receives a data item from an upstream component and sends a data item to a downstream subcomponent. On the bottom path going right to left each subcomponent receives a data item from a downstream component and sends a data item to an upstream subcomponent.

Requests containing an interpreter address and an application arrive at the scheduler. The scheduler separates the interpreter address from the application and sends only the application downstream. Next, the scheduler either looks up the application's result in the cache or sends the application downstream. Last, if the application has not been served by the cache, the scheduler associates an idle worker address with the address and sends the pair to that worker. The worker is responsible for executing that application and to find its corresponding result value.

On the way back the scheduler receives a triple consisting of the worker address responsible for the execution, the original application, and the execution result. The scheduler separates the worker address from the triple, marking the worker idle and sending the remaining application-result pair upstream. Next, the scheduler adds the result to the cache so that requests for the same application can be answered from the cache. Last, the scheduler rebinds the application-result to the interpreter address that expects its execution and sends the application-result pair as a reply to the requesting interpreter.

Herein, both workers and interpreters can fail. When a worker fails the scheduler removes it from its worker pool. If the failing worker is busy executing a foreign function application, the scheduler reschedules the application. When an interpreter fails, the scheduler drops all outstanding replies to that interpreter.

The scheduler exposes two structural decisions: An application can either be allowed for scheduling or served by the cache. Also, a failing worker can either be removed silently, or reschedule an application upon removal. However both decisions are only structural. At most one of the transitions involved in a decision is enabled for any specific application or worker.

4.4. Worker

The worker executes foreign function applications it gets from the scheduler. Typically, several workers are available to a scheduler at a time. The amount of workers available to a scheduler ranges from four to eight for a single computer to several hundred or thousand for large clusters. Each worker processes applications independent of any other worker. This way, we tap the parallelization potential in a workflow. Each worker has access to several foreign language environments, e.g., Python or R, including their attached libraries, and to a distributed file system. The distributed file system is the source of input data and the destination of output data of an application execution. This

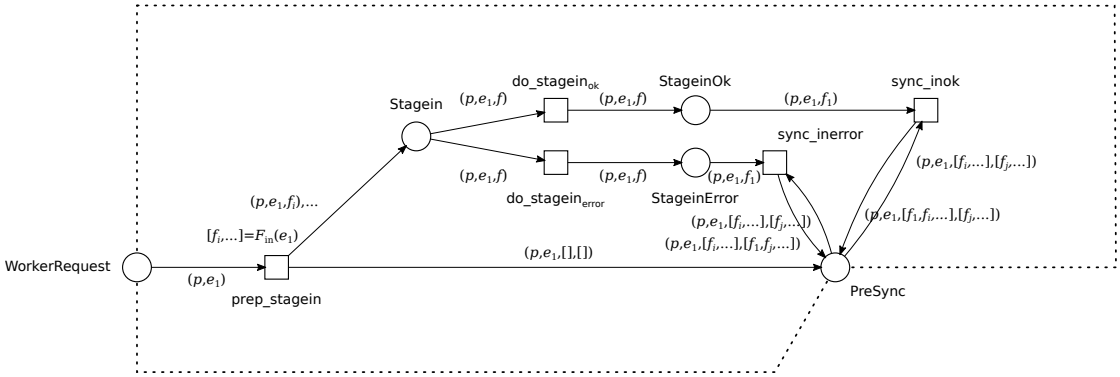


Figure 4.11.: Petri net model of the stage-in worker sub-component

means that the worker merely coordinates the data transfer and data processing while it is the distributed file system that performs the data transfer and the foreign language environment that performs the data processing.

A worker receives a request in the form of an address-application pair where the address uniquely identifies the worker. It replies with a triple containing the worker address, the original application, and the execution result. Herein, the result is either a value or an error expression.

A worker process computes the result of a foreign function application in three subsequent steps: (i) it stages in all data the foreign function application needs by copying all input files from the distributed file system, (ii) it runs the foreign function script inside a temporary directory consuming the input data and producing the output data, and (iii) it stages out all data produced the foreign function application produces by copying all output files to the distributed file system. Herein, the stage-in and stage-out steps are structurally similar.

Each of the three steps may fail. E.g., a foreign function application may expect a file that does not exist. The execution may return an error, for example because the script contains a syntax error or an exception is raised. Lastly, the script may fail to produce an output file the user expects it to produce. Assuming that the foreign function script is deterministic, rescheduling the application would result in the same failure. So, the worker propagates the error to the scheduler which, in turn, propagates the error to the requesting interpreter.

4.4.1. Data Staging

Before executing a foreign function application, the worker copies all input data to the worker machine. Conversely after execution, it copies all output data to the distributed file system. The stage-in and stage-out steps that handle data transfers from and to the distributed file system are structurally similar. First, the worker determines a list of files that need to be staged. Next, the worker attempts, independently for each file, to perform the staging operation which can either succeed or fail. When all staging

operations have been attempted the worker observes whether the all staging operations were successful. If so, the worker continues. If not, an error is propagated back to the scheduler that contains a list of all files that failed to be staged.

Figure 4.11 shows the Petri net model of the stage-in subcomponent. When a process-application pair arrives at the WorkerRequest place, the prep_stagein transition augments the pair with two empty lists: the list of successful transfers and the list of failed transfers. This tuple appears on the PreSync place. The same transition produces a token for each file to stage on the Stagein place. Processing a file can either succeed, in which case the do_stagein_ok transition forwards the token to the StageinOk place, or fail, in which case the do_stagein_error transition forwards the token to the StageinError place. Lastly, the tuple on the PreSync place successively gathers all staged files, putting each filename in either the failed list or the successful list.

4.4.2. Foreign Function Application Execution

When all input data is available on the worker machine, execution can begin. A foreign function application contains two pieces of information: (i) the foreign function, enumerating all arguments and their types, the result's type, the foreign language, and the foreign function script, and (ii) the argument bindings stating for each argument what value to bind to it. The worker uses that information to compile a script to run on the worker machine. This script is based on the foreign function script. We discuss the details of serializing foreign function applications and compiling the execution script in Section 5.5.

The execution can either be a success or fail. If the application executes successfully it produces a value that is propagated upstream. If execution fails the worker generates an error expression that it propagates to the scheduler.

4.4.3. Worker Composition

Lastly, we compose the worker subcomponents to form the worker net. The subcomponents we compose are the stage-in, execution, and stage-out subcomponent. Figure 4.12 shows the result of this composition.

The worker receives requests from the scheduler via the WorkerRequest place and sends replies via WorkerResult place. Each of the three steps has a success case which enables the next step and an error case which produces an error expression on the WorkerResult place. Consequently, for any request that arrives on the WorkerRequest place at some point in the future a token appears on the WorkerResult place. Also, there remain no residual tokens in the worker net.

In fact, The worker is the only component of Cuneiform's distributed execution environment that is a sound workflow net [228, 229, 230]. Note that the soundness property cannot be verified by looking only at the net structure since some of the transitions have guards. However, by close observation we find that the net always enables some transition if the current marking is reachable from a marking where there is exactly one token on the WorkerRequest place and no other tokens in the net.

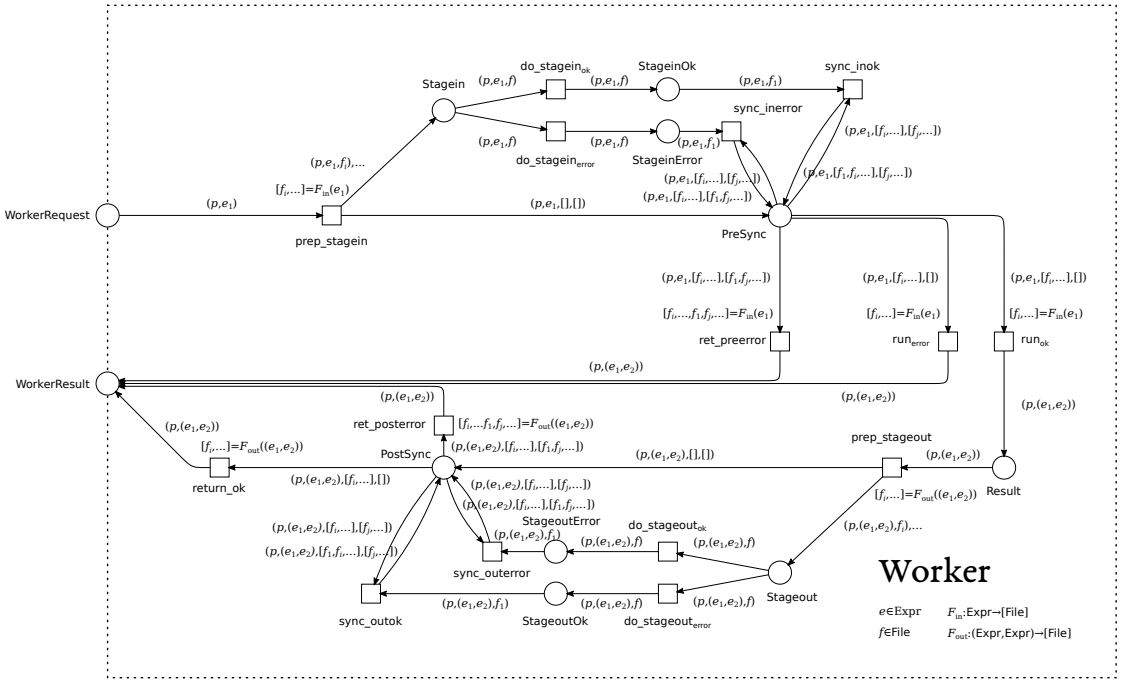


Figure 4.12.: Petri net model of the worker component. The worker composes a stage-in, execution, and stage-out subcomponent.

4.5. Further Reading

The modeling of a distributed execution environment for Cuneiform underwent several stages. At first, we did not model workflow execution at all. Nevertheless, we discuss a Hadoop-based version of a distributed execution environment for Cuneiform in Brandt et al. 2015 [31] and Bux et al. 2015 [37] without a formal specification. Later, we give a Petri net model for only the interpreter in Brandt et al. 2017 [33]. A Petri net model of the scheduler appears in Brandt and Reisig 2018 [32]. Reisig 2013 [194] provides an introduction to high-level interface nets and Petri net properties.

5. Implementation

In Chapter 3 we specify Cuneiform's abstract syntax and semantics. In Chapter 4 we specify a distributed execution environment for a Cuneiform interpreter. In this chapter we describe how we use these specifications to build a concrete Cuneiform interpreter and distributed execution environment.

Although a Java and Hadoop based implementation of an interpreter-execution environment pair exists [31, 39, 202], for the purpose of this thesis we describe an implementation based on Erlang. We implemented both the interpreter and the distributed execution environment in Erlang. In this chapter we discuss some of the important modules this implementation hosts.

In Section 5.1 we review Erlang as a platform for programming languages and distributed systems. Section 5.2 describes Cuneiform's concrete syntax which is based on the abstract syntax given in Section 3.2. Section 5.4 describes how we implement the Petri net specifications given in Chapter 4 as Erlang processes. Section 5.5 describes the foreign function interface the distributed execution environment uses to interface with other languages. The distributed execution environment produces an execution log. In Section 6.2 we give the format of this log and discuss the insights we gain from analyzing these logs. Lastly, Section 5.6 colloquially summarizes the development stages the Cuneiform implementation has gone through and the lessons we learned in the process.

5.1. Erlang

To implement a Cuneiform interpreter and distributed execution environment we translate its specification into software. The Erlang programming language brings many features for building new programming languages and distributed systems.

First, Erlang is a functional programming language with pattern matching which, in our experience, eases implementing an internal language model and semantics in comparison to, e.g., Java. The reason is that reduction semantics, which we use to give the Cuneiform semantics, is already very close to the functional programming paradigm. Also, since the reduction semantics are a variant of operational semantics, it is always obvious from the form of an expression which reduction rule to apply. Hence, we can use the pattern matching feature of Erlang to distinguish the different cases. In most object-oriented languages, like Java, Groovy, Kotlin, or Smalltalk, we have to chain getters and conditions to distinguish these cases. While the getter-condition combination is decent enough, the pattern matching approach is even more convenient.[1] Using pat-

[1]Note that there are also object-oriented languages that provide pattern matching, e.g., Scala, C#, or C++14 and above. The case for Erlang becomes more clear once we take distribution into account.

tern matching leads to a shorter and more maintainable implementation than could be achieved in a traditional object-oriented language.

In addition, Erlang allows us to compose distributed systems from independent processes that interact by passing messages. Processes can monitor or link to other processes. When a monitored process fails a message is sent to the monitoring process. Linked processes fail together. Also, distributed Erlang allows us to join multiple Erlang instances so their processes can interact. The message passing, monitoring, and linking functionality remains the same in the distributed case. Thus, Erlang provides a convenient platform for creating distributed systems. Since we give the communication patterns of the distributed execution environment as Petri nets, we need a way to get an Erlang process from a Petri net specification. This translation is readily achievable with the Petri net library gen_pnet.[2] This library provides a small domain-specific language for describing Petri nets. Then, the library enacts the specification in the guise of a conventional Erlang process. To the outside such a process sends and receives messages like any Erlang process, however, the internal behavior of that process is defined by a Petri net.

Lastly, we use several Erlang libraries to make the impedance between the reference provided in Chapters 3 and 4 and the implementation as small as possible. We use Leex[3] and Yecc to generate a parser for Cuneiform's concrete syntax. Next, we use Jsone[4] to encode and decode foreign function applications and logs as Json documents. We use the Getopt library[5] to parse command line arguments. We use the Cowboy web-server[6] to serve logs and status information. Lastly, we use the Rebar 3 build system[7] to manage dependencies.

5.2. Concrete Syntax and Parser

This section defines Cuneiform's concrete syntax. The concrete syntax defines the form of valid Cuneiform programs. It is an exhaustive specification for a scanner and a parser. Cuneiform's implementation uses this specification as an input for Leex and Yecc, two Erlang libraries constituting Erlang's default parser generator.

We motivate each syntactic category and provide a small example where appropriate. These running examples also appear in the definitions of the abstract syntax (see Section 3.2), type rules (see Section 3.3), and reduction relation (see Section 3.5). The railroad diagrams for each syntactic category are generated using the Railroad Diagram Generator[8].

[2] `https://github.com/joergen7/gen_pnet`
[3] `https://github.com/rvirding/leex`
[4] `https://github.com/sile/jsone`
[5] `https://github.com/jcomellas/getopt`
[6] `https://ninenines.eu/`
[7] `https://www.rebar3.org/`
[8] `http://bottlecaps.de/rr/ui`

5.2.1. Script

A *script* is the root of a concrete Cuneiform program. A script consists of one or more statements.

```
script       ::= statement+
```

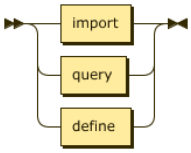
For example, the following snippet shows a script of two statements: a let definition (see Section 5.2.5) and a query (see Section 5.2.2).

```
let x : Str = "blub";
x;
```

5.2.2. Statement

A Cuneiform script consists of *statements*. A statement can be either an import, a query, or a definition.

```
statement    ::= import
               | query
               | define
```



For example, the first line of the previous snippet, the let definition (see Section 5.2.5), is a statement.

```
let x : Str = "blub";
```

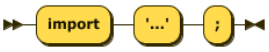
The second line of the previous snippet, is a query statement.

```
x;
```

5.2.3. Import

An *import* statement allows the user to import an external Cuneiform source file. It consists of the keyword `import`, a single-quoted filename, and a terminating semicolon. Importing replaces the import statement with all definitions in the designated source file.

```
import       ::= 'import' "'...'" ';'
```



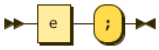
For example, the following statement imports the file `std.cfl`.

```
import 'std.cfl';
```

5.2.4. Query

Neither imports nor definitions (see Section 5.2.5) trigger any computation. Instead, the Cuneiform interpreter collects these statements until the user enters a *query* statement. A query is an expression (see Section 5.2.6) terminated by a semicolon and is interpreted in the context of all previous definitions.

```
query       ::= e ';'
```



For example, the second line of the script example (see Section 5.2.1) is a query. It queries the variable `x`.

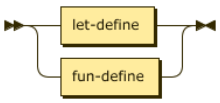
```
x;
```

5.2.5. Definition

A *definition* binds an expression to a variable name. Cuneiform has two kinds of definitions: a function definition binds a function to a variable name; a let definition binds a non-function expression to a variable name. In Cuneiform, like in Scheme, functions and other expressions inhabit the same variable namespace. Variables are lexically scoped. Redefining a variable shadows all previous definitions.

In contrast to languages with a Call-by-Value evaluation strategy, a Cuneiform definition never triggers computation. Instead, the Cuneiform interpreter collects definitions until it encounters a query. Then it evaluates the query inside the closure holding all previously recorded definitions.

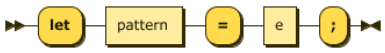
```
define      ::= let-define
              | fun-define
```



Let Definition

A *let definition* binds an expression to a variable name. It consists of the keyword `let`, a pattern, an equals sign, the expression to be bound, and a terminating semicolon. The Cuneiform interpreter unifies the expression on the right-hand side of the equals sign with the pattern on the left-hand side. The interpreter matches all variable names appearing in the pattern with their corresponding sub-expressions in the bound expression. The type system determines upfront whether a match is possible, so pattern matching never fails at runtime.

```
let-define   ::= 'let' pattern '=' e ';'
```



For example, the first line of the script example (see Section 5.2.1) is a let definition binding the variable `x` of the type `Str` to the string literal `"blub"`.

```
let x : Str = "blub";
```

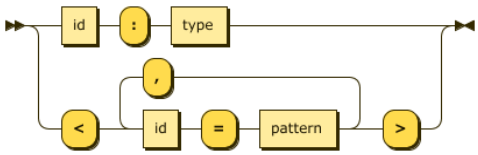
In a second example we bind the variable `y` of the type `Bool` by accessing a record field `a` using pattern matching. The right-hand side of the let definition is a record literal (see Section 5.2.6).

```
let <a = y : Bool> =
  <a = true, b = "blub">;
```

Pattern

A *pattern* appears in a let definition to determine which variable name should be bound, what type that variable has, and how to access the data structure on the right-hand side of a let definition to extract its value. A pattern can be either a name-type pair or a record pattern with one or more fields each associated to a pattern.

```
pattern      ::= id ':' type
               | '<' id '=' pattern (',' id '=' pattern)* '>'
```



For example, the name-type pair on the left-hand side of the first let definition example has the following pattern:

```
x : Str
```

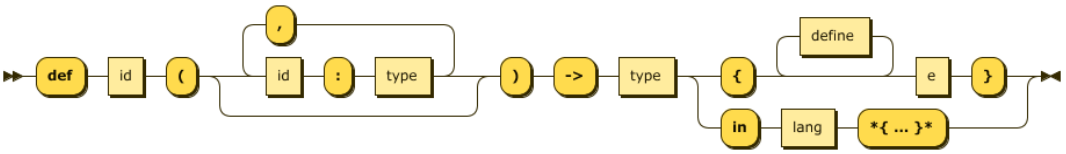
The the second let definition example has the following pattern:

```
<a = y : Bool>
```

Function Definition

A *function definition* binds a function to a variable name. A function starts with the keyword `def` and has a name, a list of typed arguments, a return type, and a body. Cuneiform has two types of functions: native functions and foreign functions. Both types differ in the form of their body. A native function body consists of a series of definitions followed by an expression. A foreign function body has a language identifier and a body string enclosed in Mickey mouse-eared curly braces.

```
fun-define   ::= 'def' id '(' (id ':' type (',' id ':' type)* )? ')'
                 '->' type ( 'in' lang '*{ ... }*' | '{' define* e '}' )
```



For example, in the following we define the function `samtools-sort`. It takes a file argument `bam` and returns a record with a single field `sorted`, that is also a file. The function body is given in Bash.

```
def samtools-sort( bam : File ) ->
      <sorted : File>

in Bash *{
  sorted=sorted.bam
  samtools sort -m 2G $bam -o $sorted
}*
```

In a second example we define a native function `forall` that checks if all elements in a list are true. It takes a single argument `l` which is a Boolean list and returns a Boolean. In its body we fold over the list, using the `and` operator.

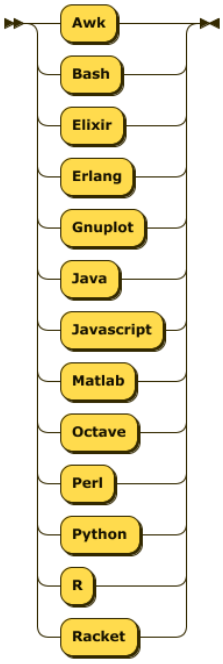
```
def forall( l : [Bool] ) -> Bool

{
  fold acc : Bool = true, x : Bool <- l do
    (acc and x)
  end
}
```

Foreign Language

A *foreign language* identifier is a keyword that appears in a foreign function definition.

```
lang         ::= 'Awk'
               | 'Bash'
               | 'Elixir'
               | 'Erlang'
```

```
  | 'Gnuplot'
  | 'Java'
  | 'Javascript'
  | 'Matlab'
  | 'Octave'
  | 'Perl'
  | 'Python'
  | 'R'
  | 'Racket'
```

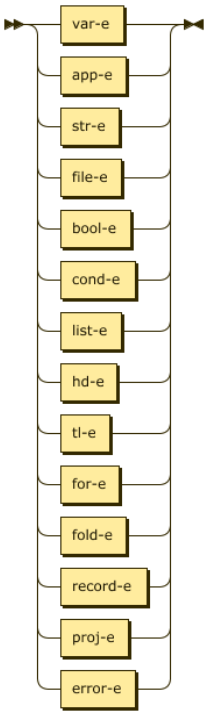


5.2.6. Expression

Expressions are the main building blocks of Cuneiform programs. Expressions can be composed to larger expressions. They appear in queries (see Section 5.2.2) and definitions (see Section 5.2.5).

```
e             ::= var-e
                | app-e
                | str-e
                | file-e
                | bool-e
                | cond-e
                | list-e
                | hd-e
                | tl-e
                | for-e
                | fold-e
```

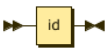
```
| record-e
| proj-e
| error-e
```



Variable

A variable expression is a placeholder for an expression.

```
var-e       ::= id
```



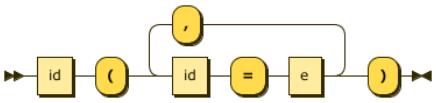
For example, the query statement in the second line of the script example (see Section 5.2.1) contains a variable expression.

```
x
```

Function Application

A *function application* consists of a function expression and an argument binding list. The interpreter substitutes all occurrences of an argument variable in the body of that function with the bound expression.

```
app-e       ::= id '(' id '=' e (',' id '=' e)* ')'
```



For example, the following snippet shows a function application where we apply the function bound to the variable `forall`. Herein, we bind the argument `l` to a Boolean list literal.

```
forall( l = [true, false : Bool] )
```

String

A *string* literal is a sequence of characters enclosed in double quotes.

```
str-e        ::= '"..."'
```

For example, the following snippet shows a string literal:

```
"blub"
```

File

A *file* literal is a sequence of characters enclosed in single quotes designating an existing file. When the user applies a function with a file as an argument, the file is staged-in to the machine where the scheduler assigns the function application to run.

```
file-e       ::= "'...'"
```

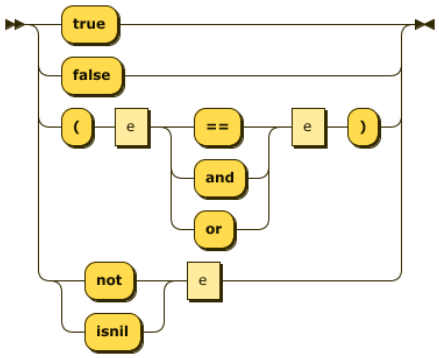
For example, the following snippet shows a file literal:

```
'file.txt'
```

Boolean Expressions

A *Boolean* expression stands for either true or false. In addition to Boolean literals Cuneiform provides comparison, negation, conjunction, disjunction, and nil-test operations.

```
bool-e       ::= 'true'
               | 'false'
               | '(' e '==' e ')'
               | 'not' e
               | '(' e 'and' e ')'
               | '(' e 'or' e ')'
               | 'isnil' e
```



For example, the following expression compares two strings.

```
("bla" == "blub")
```

The following expression combines several Boolean operations on the variables `a` and `b`:

```
((not a or not b) == not (a and b))
```

The following expression tests whether a string list is empty:

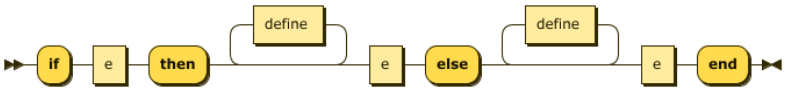
```
isnil ["bla", "blub" : Str]
```

Conditionals

A *conditional* allows the user to branch execution depending on a condition to be either true or false. The conditional expression starts with the keyword `if` followed by an Boolean expression. The keyword `then` marks the beginning of the then-block; the keyword `else` marks the beginning of the else-block. Both blocks are a series of definitions followed by an expression. The keyword `end` marks the end of the conditional expression.

Many programming languages introduce conditionals as control structures that guard the execution of their branches but are no expressions in their own right. In contrast, Cuneiform conditionals (like conditionals in functional languages like Scheme) are expressions, i.e., they represent a value.

```
cond-e       ::= 'if' e 'then' define* e 'else' define* e 'end'
```



For example, the following conditional trivially evaluates to its then-branch.

```
if true then "bla" else "blub" end
```

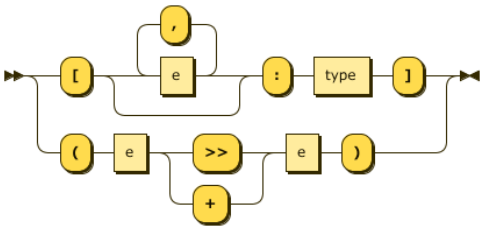
In another example we compare the string stored in the variable `a` with the string `"ok"`. In the then-branch we return the variable `x`. In the else-branch we call the function `f` on `x`.

```
if( a == "ok" )
then
  x
else
  f( x = x )
end
```

Lists

A *list* expression is an ordered collection of expressions that have the same type. Lists are one of two composite data structures in Cuneiform. Cuneiform provides three ways to construct a list: First, a list literal enumerates the list's elements inside squared brackets. Each element of the list literal is checked against the declared type at the end of the enumeration. Next, a construction expression prefixes an existing list with an element. Last, an append expression concatenates two lists.

```
list-e       ::= '[' (e (',' e)*)? ':' type ']'
               | '(' e '>>' e ')'
               | '(' e '+' e ')'
```



For example, the following expression is a string literal enumerating Boolean values.

```
[true, false, false : Bool]
```

The next expression constructs the same list but starts with a literal empty list and prefixes it three times.

```
( true >> ( false >> ( false >> [: Bool] ) ) )
```

List Accessors

The list accessors Cuneiform provides are *hd* and *tl* for a list's head and tail respectively. We need to be pay attention to the case where the user applies a list accessor to an empty list. Conventional programming languages type a list accessor for any operand list even though the empty list is a possibility. In the case of the empty list they throw a runtime exception. Another possibility is to derive (or let the user derive) a proof that the operand list cannot be empty. Both extensions are rather complex, and thus we go with the third option: A list accessor takes two operands: the operand list and an alternative expression that the accessor evaluates to in case the operand list is empty.

```
hd-e          ::= 'hd' e 'default' e
```


```
tl-e          ::= 'tl' e 'default' e
```

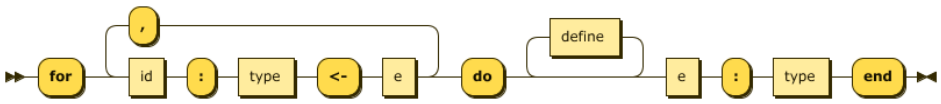
For Iteration

A *for* expression allows to iterate over a list in parallel. The for expression's head starts with the keyword `for` followed by a sequence of variable name-type-list expression triples. The for expression's body starts with the keyword `do` followed by a sequence of definitions and a body expression. The body ends with a type annotation and the keyword `end`.

Iterating over a list using a for expression creates an independent variant of the body expression for each element in a list. Consequently, the Cuneiform execution environment can parallelize foreign function applications in a for expression's body.

Specifying multiple lists in the for expression's head combines these lists element-wise, much like Common Lisp's `mapcar`, Racket's `for/list`, or Erlang's `zipwith`.

```
for-e         ::= 'for' id ':' type '<-' e (',' id ':' type '<-' e)*
                  'do' define* e ':' type 'end'
```



For example, the following expression iterates over a Boolean list literal negating each element in the list.

```
for x : Bool <- [true, false, false : Bool] do
  not x : Bool
end
```

In a second example, we iterate over a list of files in the variable `fastq1-lst-gz` calling the function `gunzip` on each of its elements.

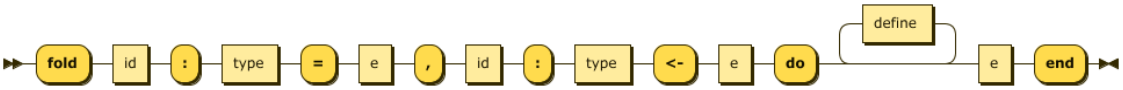
```
for gz : File <- fastq1-lst-gz do
  ( gunzip( gz = gz )|file ) : File
end
```

Fold Iteration

A *fold* expression iterates over a list altering an accumulator for each element of a list. The expression reduces to the final value of the accumulator. A fold expression's head starts with the keyword `fold` followed by the initial accumulator definition as a variable name-type-expression triple and the list to fold over, as a variable name-type-list expression triple. The fold expression's body starts with the keyword `do` followed by a sequence of definitions and an expression. The body closes with the `end` keyword.

Like a for expression, iterating over a list using a fold expression creates an independent variant of the body expression for each element in a list. Consequently, the Cuneiform execution environment can parallelize foreign function applications in a fold expression's body.

```
fold-e       ::= 'fold' id ':' type '=' e ',' id ':' type '<-' e
                 'do' define* e 'end'
```



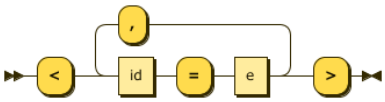
For example, the following fold iteration reverses a list by reconstructing it in the accumulator.

```
fold acc : [Str] = [: Str],
     x : Str <- ["a", "b" : Str]
do
  (x >> acc)
end;
```

Record

A *record* expression is a data structure that consists of one or more named fields. A record literal is a sequence of name-expression pairs in angle brackets.

```
record-e     ::= '<' id '=' e (',' id '=' e)* '>'
```



For example, the following expression is a record literal with a string field `a` and a Boolean field `b`.

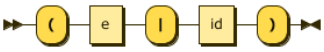
```
<a = "ok", b = true>
```

Record Projection

A *record projection* accesses a record expression. It has the form of a `|` infix operator with a record expression on the left-hand side and a field name on the right-hand side.

Cuneiform provides two ways to access records: (i) a record projection and (ii) a let definition with a record pattern. Internally, the Cuneiform parser desugars such a let definition to a record projection.

```
proj-e       ::= '(' e '|' id ')'
```



For example, the following expression projects the field `b` from a record literal.

```
( <a = "ok", b = true>|b )
```

Error

A user-defined *error* expression introduces a runtime error. It starts with the keyword `error` followed by an error message string and a type. When an error appears in the control string of the interpreter it halts execution and reports the error message to the user.

Cuneiform types a user-defined error although it is neither a value, nor can the interpreter make progress on its evaluation. So, an error circumvents Cuneiform's type system. It is a more direct way to insert a runtime error than to call a failing foreign function.

```
error-e      ::= 'error' '"..."' ':' type
```

For example, the following expression introduces an error expression in lieu of a Boolean expression inside a negation:

```
not error "kaboom" : Bool
```

The next example introduces an error on the right hand side of a disjunction.

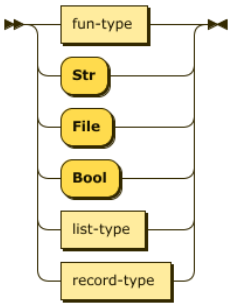
```
(true or error "kaboom" : Bool)
```

Both examples pass type checking since both errors type as a Boolean expression as expected.

5.2.7. Type

Let and Function definitions, and some expressions like lists, for and fold iterations, or errors contain *type* annotations. The Cuneiform interpreter uses these type annotations to check whether a program is consistent before running it. Cuneiform provides six types: A function type, Three base data types for strings, files, and Booleans, and two composite data types for lists and records.

```
type         ::= fun-type
               | str-type
               | file-type
               | bool-type
               | list-type
               | record-type
```

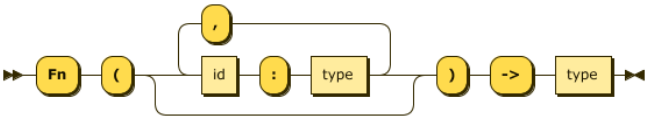


Function Type

A *function type* describes a function. While Cuneiform has native functions and foreign functions, the type system does not distinguish these two kinds of functions, i.e., a user can use a foreign function wherever he can use a native function and vice versa. A function type contains a name-type pair for each argument followed by a return type.

The syntax description below allows a function to consume and produce data of any type. However, Cuneiform's foreign function interface supports arguments only if they are base data types or lists of base data types. Also a foreign function's return type must be a record. The record fields can, again, be only base data types or lists of base data types. Cuneiform enforces these restrictions upfront as part of its type checking.

```
fun-type     ::= 'Fn' '(' (id ':' type (',' id ':' type)* )? ')'
                 '->' type
```



For example, the following type annotation describes a foreign function that consumes a string argument `person` and produces a record with a field `out` which is also a string.

```
Fn( person : Str ) -> <out : Str>
```

String Type

The *string type* is a base data type designating a character sequence.

```
str-type    ::= 'Str'
```

File Type

The *file type* is a base data type designating a character sequence that represents a file.

```
file-type    ::= 'File'
```

Boolean Type

The *Boolean type* is a base data type designating the values `true` and `false`.

```
bool-type    ::= 'Bool'
```

List Type

A *list type* is a composite data type designating a list whose elements have a defined type.

```
list-type    ::= '[' type ']'
```

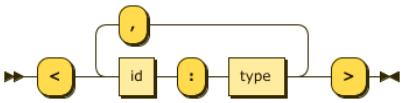
For example, the following type annotation designates a string list.

```
[Str]
```

Record Type

A *record type* is a composite data type designating a record whose fields are associated with a defined type.

```
record-type  ::= '<' id ':' type (',' id ':' type)* '>'
```



For example, the following type annotation designates a record with two fields: a string field `a` and a Boolean field `b`.

```
<a : Str, b : Bool>
```

5.3. Type System

In Section 3.3 we introduce a type system based on simple types. This type system is capable of catching many possible errors upfront. Catching errors early is important since a workflow may run for days and weeks. There are however three classes of errors that the type system cannot protect against: (i) errors explicitly triggered by the user, (ii) errors that surface while executing a foreign function application, and (iii) errors the type system does not model, e.g., empty files, files in the wrong format, etc. Below, we give a table that lists the inconsistencies the type system catches at compile time.

Error type	**Explanation**
unbound var	user dereferences an undefined variable
ntv fn ambiguous arg name	function with non-distinct argument names
frn fn ambiguous arg or return field name	foreign function with non-distinct argument or return field names
frn fn returns no rcd	foreign functions that return no record
awk frn fn first arg no file	Awk foreign function that does not process a file
awk frn fn no arg	Awk foreign function that does not process anything
awk frn fn result field no file	Awk foreign function whose `result` field is no file
awk frn fn no result field	Awk foreign function whose return record has no `result` field
app lhs no function	function application with no function in the function position
app missing bind	function application that misses at least one argument binding
app dangling bind	function application that has at least one extra argument binding
app bind type mismatch	function application where the binding has a different type than the function spec requires
app arg name mismatch	function application where the argument name in the function spec and the argument binding differ
fix fn no arg	fixpoint whose operand function has no arguments

fix fn arg type mismatch	fixpoint whose operand function's first argument does not match the recursive function's spec
fix no fn	fixpoint whose operand is no function
fix return type mismatch	the function that is the operand of the fixpoint operator has a function as its first argument but its return type differs from the type of the function's body
fix fn arg no fn	the function that is the operand of the fixpoint operator has a first argument but it is not a function
cmp no comparable type	comparison whose left-hand side cannot be compared, e.g., a function
cmp incomparable	comparison whose operands cannot be compared, e.g., a string and a Boolean
conj lhs no bool	conjugation whose left-hand side is no Boolean
conj rhs no bool	conjugation whose right-hand side is no Boolean
disj lhs no bool	disjunction whose left-hand side is no Boolean
disj rhs no bool	disjunction whose right-hand side is no Boolean
neg no bool	negation whose operand is no Boolean
isnil no list	nil-test operand is no list
cnd result type mismatch	then- and else-branches of a condition are not of the same type
cnd case no bool	case of a condition is no Boolean
cons element type mismatch	list constructor's left-hand side does not match the element type of the right-hand side
cons no list	list constructor's right-hand side is no list
hd type mismatch	head operand's element type does not match the default expression's type
hd no list	the head operand is no list
tl type mismatch	tail operand's type does not match the default expression's type
tl no list	tail operand is no list
append lhs no list	append left-hand side is no list
append rhs no list	append right-hand side is no list
append element type mismatch	append left-hand side element type mismatches right-hand side element type
for ambiguous bind name	for iteration with ambiguous iterator variables

for bind type mismatch	for iteration where an iterator variable's declared type mismatches the element type of the list to be iterated
for bind no list	for iteration attempting to iterate something that is no list
for body type mismatch	for iteration's body does not match declared type
fold ambiguous bind name	fold iteration where the accumulator name and name of the iterator variable are the same
fold acc bind type mismatch	fold iteration where the accumulator's initial value mismatches its declared type
fold list bind type mismatch	fold iteration where the iterator variable's declared type mismatches the element type of the list to be iterated
fold list bind no list	fold iteration attempting to iterate something that is no list
fold body type mismatch	fold iteration where the body's type mismatches the accumulator's type
rcd ambiguous field name	record with ambiguous field names
proj field missing	projection of a record that does not have the projected field
proj no record	projection whose operand is no record

The type system does not infer types. Also, all types are disjunct. Omitting these possibilities requires the user to provide specific type information and to form and document expectations. But also, it makes the implementation of the type system straightforward. Nevertheless, the type system is flexible enough to be useful in the context of scientific workflows. We demonstrate the usefulness of the type system in Chapter 6.

5.4. Erlang Processes from Petri Nets

Partitioning systems into independent, distributed components has many advantages but also brings about design challenges. A distributed application needs to cope with partial failure and protocols need to maintain invariants as well as liveness or termination properties which are seldom self-evident except in simplistic scenarios. Also, distributed systems are rarely deterministic which adds a layer of complexity promoting transient failures which are hard to reproduce. Distribution complicates logging and debugging since no single component can hold a consistent view on the whole application. These challenges force us to design the components and protocols of a distributed system with extra care. The OTP framework offers several behaviors to support the design of processes.

More and more software applications operate in a distributed setting. The fledging fields of service orientation [174, 116, 177] and microservices [60] opened up a new class of

systems and come with their own design challenges. Partitioning a software application into independent services allows for a stricter modularization than is possible in closely coupled applications. This modularization unlocks a computational speedup through parallelization and the opportunity to use different software stacks provided they share an interface to communicate.

Erlang and the Open Telecom Platform (OTP) [220] framework address the challenges of distribution by providing process templates that separate application-dependent from application-specific behavior [42]. Accordingly, the OTP framework provides various process templates. The OTP is a collection of libraries for Erlang. An OTP *behavior* is a design convention, frequently applied in Erlang libraries. It combines (i) a collection of functions acting as an application-independent scaffold for a process, controlling its life cycle and communication interface, and (ii) a collection of callback specifications which outline the application-specific parts of that process. Important examples are the generic server, event handler, supervisor, or finite state machine behaviors.

Researchers apply Petri nets to model biological systems [104], algorithms, or communication protocols. They are especially suited for modeling distributed software systems because they explicitly specify choice, dependence and independence, which are central to the understanding of distribution. A Petri net makes progress in discrete transitional steps. Because Petri nets allow steps to occur independently across one net these steps are, generally, unordered. Also, a transitional step assumes no knowledge about the system other than what directly enables it.

The independence of transitional steps in a Petri net mirrors the independence of events in distributed systems in which, generally, the order of events can be neither observed nor enforced. And the restriction of a transitional step to affect only a closed and predefined set of adjacent components mirrors the fragmentation of distributed systems in which, generally, the state of the system as a whole can be neither observed nor enforced. Consequently, Petri nets are an ideal modeling framework for distributed systems.

Here, we introduce gen_pnet[9], a behavior for deriving Erlang processes from Petri nets. We give a short introduction to Petri net semantics and demonstrate how we can model applications as Petri nets and execute them on an Erlang virtual machine. Gen_-pnet allows specifying interface place-transition nets with arbitrary data. I.e., the net appears to the outside as an Erlang process that sends and receives Erlang messages (the interface). To the inside the process consists of places that hold Erlang data structures and transition that consume and produce Erlang data structures. This is similar to the way a gen_fsm appears to the outside as an ordinary Erlang process while its internal behavior is defined as a finite state machine.

Herein, the gen_pnet OTP behavior extends the gen_server behavior to allow seamless integration into Erlang applications: Since a gen_pnet instance is a process like any other it can send and receive messages, can be called, cast on, monitored, linked to, supervised, or handle hot code reloading.

We define Petri nets by creating a callback module that implements the gen_pnet

[9] `https://github.com/joergen7/gen_pnet`

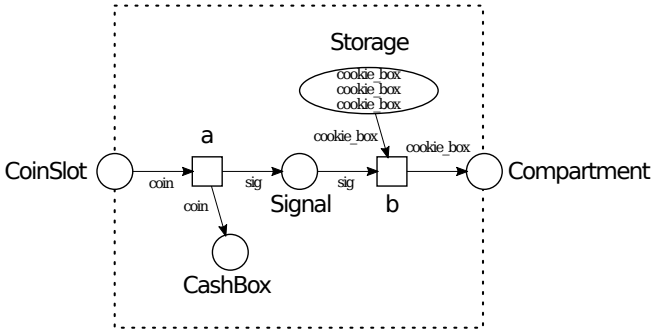


Figure 5.1.: Petri net model of a cookie vending machine

behavior providing the following callback functions.

`place_lst/0` returns the names of the places in the net.

`trsn_lst/0` returns the names of the transitions in the net.

`init_marking/2` returns the initial marking for a given place.

`preset/1` returns the preset places of a given transition.

`is_enabled/2` determines whether a given transition is enabled in a given mode.

`fire/3` returns which tokens are produced on what places if a given transition is fired in a given mode that enables this transition.

`trigger/3` allows to add a side effect to the generation of a token.

In addition, the gen_pnet behavior defines some more callback functions like `code_change/3` or `handle_info/2` which it inherits from the gen_server behavior it extends.

5.4.1. Example: Cookie Vending Machine

We exemplify the implementation of a Petri net with gen_pnet by implementing the cookie vending machine depicted in Figure 5.1. For the sake of simplicity this cookie vending machine model provides neither a way to refill the storage nor to empty the cash box since both places are inaccessible via the interface.

Next, we define each callback function in turn. The `place_lst/0` function lets us define the names of all places in the net. Here, we define the net to have the five places in the cookie vending machine.

```
place_lst() ->
  ['CoinSlot', 'CashBox', 'Signal', 'Storage',
   'Compartment'].
```

The `trsn_lst/0` function lets us define the names of all transitions in the net. Here, we define the net to have the two transitions a and b.

```
trsn_lst() -> [a, b].
```

The `preset/1` function lets us define the preset places of a given transition. As its argument it takes the transition's name. Here, we define the preset of the transition a to be just the place CoinSlot while the transition b has the places Signal and Storage in its preset.

```
preset( a ) -> ['CoinSlot'];
preset( b ) -> ['Signal', 'Storage'].
```

The `init_marking/2` function lets us define the initial marking for a given place in the form of a token list. As arguments it takes a place name and a user info field. Here, we initialize the storage place with three `cookie_box` tokens. All other places are left empty.

```
init_marking( 'Storage', _UsrInfo ) ->
  [cookie_box, cookie_box, cookie_box];

init_marking( _Place, _UsrInfo ) ->
  [].
```

The `is_enabled/3` function is a predicate determining if a given transition is enabled in a given mode. As arguments it takes a transition name, a firing mode in the form of a hash map mapping place names to token lists, and a user info field. Here, we state that the transition a is enabled if it can consume a single `coin` from the CoinSlot place. Similarly, the transition b is enabled if it can consume a `sig` token from the Signal place and a `cookie_box` token from the Storage place. No other configuration can enable a transition. E.g., managing to get a `button` token on the CoinSlot place will not enable any transition.

```
is_enabled( a, #{ 'CoinSlot' := [coin] },
            _UsrInfo ) ->
  true;

is_enabled( b, #{ 'Signal'  := [sig],
                  'Storage' := [cookie_box] },
            _UsrInfo ) ->
  true;

is_enabled( _Trsn, _Mode, _UsrInfo ) ->
  false.
```

The `fire/3` function defines what tokens are produced when a transition fires in a given mode. Like `is_enabled/3` it takes as arguments a transition name, a firing mode, and

a user info field. The `fire/3` function is called only on modes for which `is_enabled/2` returns `true`. The `fire/3` function is expected to return either a {`produce`, `ProduceMap`} tuple or the atom `abort`. If `abort` is returned, the firing is canceled, i.e., nothing is produced or consumed. Here, the firing of the transition `a` produces a `coin` token on the CashBox place and a `sig` token on the Signal place. Similarly, the firing of the transition `b` produces a `cookie_box` token on the Compartment place. We do not need to match in the function head the tokens to be consumed because the firing mode already uniquely identifies these tokens.

```
fire( a, _Mode, _UsrInfo ) ->
  {produce, #{ 'CashBox' => [coin],
               'Signal'  => [sig] }};

fire( b, _Mode, _UsrInfo ) ->
  {produce, #{ 'Compartment' => [cookie_box] }}.
```

The `trigger/3` function determines what happens when a token is produced on a given place. As arguments it takes a place name, the token about to be produced, and a user info field. The `trigger/3` function is expected to return either `pass` in which case the token is produced normally, or `drop` in which case the token is forgotten. Here, we simply let any token pass.

```
trigger( _Place, _Token, _UsrInfo ) -> pass.
```

Taken together, the callback functions that make up the gen_pnet behavior form a domain-specific language that allow to specify a Petri net in terms of Erlang functions. We use gen_pnet to implement the Cuneiform interpreter and distributed execution environment as we specify it in Chapter 4.

5.5. Erlang Foreign Function Interface

Evaluating a program the Cuneiform interpreter generates independent foreign function applications. The Cuneiform scheduler selects a Cuneiform worker process to execute this foreign function application. The Cuneiform worker, in turn, stages in all input files so that the application's pre-conditions are met. Next, it prepares the foreign function script, binding all input variables and appending code that outputs the output variables. The worker starts the foreign function script using the appropriate foreign language runtime environment. Afterwards, the worker collects the script's output. Then, it checks if all post-conditions are met and stages out the foreign function application's output files. Eventually, the Cuneiform worker submits the application's result to the scheduler which relays the reply to the Cuneiform client.

A foreign function application generated by the Cuneiform interpreter has a specific format. Also, the reply the interpreter expects from the scheduler has a specific format. Preparing the foreign function script, executing it, and collecting the output varies for

each supported foreign programming language. Thus, we subsumed this functionality in a library: the Erlang foreign function interface (Effi)[10].

```
{ "app_id": "1234",
  "lambda": { "lambda_name":  "bowtie2-build",
              "arg_type_lst": [{ "arg_name": "fa",
                                 "arg_type": "File",
                                 "is_list":  false }],
              "ret_type_lst": [{ "arg_name": "idx",
                                 "arg_type": "File",
                                 "is_list":  false }],
              "lang":         "Bash",
              "script":       "bowtie2-build $fa bt2idx\nidx=idx.tar\ntar cf $idx --remove-files bt2idx.*\n" },
  "arg_bind_lst": [{ "arg_name": "fa",
                     "value":    "chr22.fa" }] }

{ "app_id": "1234",
  "result": { "status":       "ok",
              "stat":         { "run":  { "t_start":  "1523007609917834743",
                                          "duration": "30391761645" },
                                "node": "cf_worker@x240" },
              "ret_bind_lst": [{ "arg_name": "idx",
                                 "value":    "idx.tar" }] } }
```

5.6. Experiences

Cuneiform has been implemented several times based on various programming platforms and using various modeling methods. In this section we revisit the implementation cycles the Cuneiform project went through. In the first cycle we used no modeling methods at all. At present, Cuneiform implements a language model based on reduction semantics and a distribution model based on Petri nets. In each implementation cycle we encountered a limit in the feature set we could deliver with justifiable complexity. Each time, we overcame this limit by improving our modeling techniques, simplifying the implementation and, thereby, making room for more features in a code base of similar size. Accordingly, Cuneiform started out with 19300 lines of Java code in its first public release[11] shrinking to 9200 lines of Erlang code in its current release[12] including test code and excluding external dependencies.

Starting out, we modeled only Cuneiform's concrete syntax in extended Backus-Naur form using an Antlr[13] script. We implemented the the semantic model and the execution environment ad-hoc. We created two independent execution environments: (i) a multi-threaded implementation running on a single machine for testing and (ii) a distributed implementation running on a cluster using Hadoop[14]. Unlike later drafts, this

[10] https://github.com/joergen7/effi
[11] https://github.com/joergen7/cuneiform/releases/tag/2.0.0-beta
[12] https://github.com/joergen7/cuneiform/releases/tag/3.0.4
[13] https://www.antlr.org/
[14] https://github.com/marcbux/Hi-WAY

implementation's interpreter *traverses* a static workflow graph, instead of *reducing* an expression.

We switched to a reduction-based interpreter in a second Java-based implementation. We integrated conditionals into Cuneiform but failed to add recursion. A mismatch in the reduction and substitution schemes caused the interpreter to go into an infinite substitution loop whenever a user applied recursion.

We overcame this limitation by creating the first draft of a structural operational semantics resulting in the third Java-based implementation of Cuneiform. This implementation featured an interpreter that used two threads: an interpreter thread that reduced the Cuneiform program and a relay thread that sent independent function applications to either a local thread-pool or a Hadoop application master and sent the results back to the interpreter thread. The interpreter and the relay thread communicated via shared mutable data structures. When one of the thread pair died it would leave an orphaned partner behind. Also we were unable to exclude the possibility of a race condition in the access of the shared data structures the threads used to communicate.

We migrated the code base to Erlang resulting in the first Erlang-based implementation. This allowed us to adopt Erlang's process model avoiding a shared state. Also, adopting Erlang enabled us to link or monitor processes and, thus, to react to partial failure.

We drafted and rolled out several cycles of refining our models and re-implementing them. We added a simple type system, found a way to generate Erlang modules from Petri net specifications, and adopted distributed Erlang which allowed us to drop the distinction between local and distributed execution environments.

Looking back, we profited from classic programming language theory and distributed systems. Also, our switch to Erlang permitted us more implementation cycles to gather more practical experience and to adapt our models accordingly.

6. Applications

Judging the aptitude of a language is hard because a survey investigating the user experience is outside the scope of this thesis. Nevertheless, we evaluate whether Cuneiform is comprehensive enough for its application domain and whether it is reliable enough to run in a distributed environment, managing many computers for a long time period. [202] shows how to port a variant calling workflow to Cuneiform and run it in a Hadoop-based distributed execution environment.

In this chapter we introduce workflow applications in which we explore the aptitude of Cuneiform. We perform this informal evaluation for five different applications from next-generation sequencing and bioinformatics.

6.1. Variant Calling using VarScan

As the first application we present a variant calling workflow. Variant calling is especially useful to show how Cuneiform parallelizes and distributes independent foreign function applications. In variant calling we typically analyze several Giga Bytes of data. Also, we split the data into hundreds of partitions each to be analyzed independently. The workflow consists of several subsequent analysis steps entailing quality control, alignment, sorting, merging, multiple pileup, variant calling, and annotation. We show that we can fully saturate a cluster of eight workstations amounting to 576 compute cores. This cluster spends about 5% of its time idle while waiting for data transfers to and from its distributed file system residing on four connected commodity PCs.

A common problem in next-generation sequencing is to identify variants in a genomic sequence sample. A variant is a difference between a sequence sample and a genomic reference. Different types of variants exist, e.g., a single-nucleotide polymorphism is a difference in just one nucleotide position. An indel is the insertion or deletion of a small number of nucleotides. Herein, single-nucleotide polymorphisms and indels are easy to detect which is why they are the most commonly observed variant. Another type of variant, a copy number variation, is a difference in the number of occurrences of a repetitive sequence. A third type of genetic variant is a translocation, which is a genetic region that appears in a different place in the sample than in the reference. Additionally, variants can be either *germline variants*, i.e., variants that appear consistently in each cell of the organism, or *somatic variants*, i.e., variants which appeared spontaneously. Somatic variants are present only in some cells of the organism, e.g., in a tumor. Thus, when detecting somatic variants it is necessary to provide either a second sample or a collection of known variants for comparison.

6.1.1. Methods

We present a Cuneiform program for detecting germline single-nucleotide polymorphisms and indels using the variant caller VarScan [132, 133, 134]. In addition, we use ANNOVAR [235, 236] to annotate the called variants. Annotation looks up the variant in a database to identify its meaning. Furthermore, ANNOVAR identifies a variant as either silent, appearing in an open reading frame, or introducing a premature stop codon. Prior to variant calling, the sequence sample must be aligned to a reference genome for which we use the alignment tool Bowtie 2 [137]. The resulting alignments are processed resulting in a multiple pileup format using SAMtools [143]. To verify the quality of the sequence sample we use FastQC [1].

The input data for the Cuneiform program consists of a sequence sample, a reference genome, and an annotation database. The sequence sample is a 98 GiByte collection of short DNA reads in FastQ format. The Human reference genome we use has a size of 3 GiByte and is provided in FastA format. Lastly, the annotation database we use with ANNOVAR has a size of 235 MiByte. Thus, in total, the program processes around 101 GiByte of data.

6.1.2. Results

First, we align the sequence sample to the reference genome using Bowtie 2. This alignment step requires an index of the reference genome which we also create using Bowtie 2. The alignment step produces a collection of alignment positions in BAM format. These alignments are sorted and a multiple pileup table is created using SAMtools. The pileup table is fed to VarScan which produces a variant collection in VCF format. Lastly, the variant collection is annotated using ANNOVAR, which results in a CSV table listing for each variant, where it is located in the genome, and what its presumed meaning is. ANNOVAR derives these annotations either from a database comprising knowledge about the consequences of a variant in a certain location or from our knowledge about the gene transcription process. Examples for such an annotation are an altered amino acid position or a premature stop-codon. Independently, the genetic sample undergoes statistical quality control using FastQC.

6.1.3. Discussion

The variant calling workflow consists of several consecutive steps. Out of these steps it is the read mapping step that is the most costly. I.e., in this processing step we spend the most CPU time. In principle, each read can be aligned to the reference genome independently. Thus, the read mapping step can be scaled almost without limit by splitting up the input data before processing it. This way, we can saturate a medium-size cluster using Cuneiform.

[1] `https://www.bioinformatics.babraham.ac.uk/projects/fastqc/`

6.2. Execution Logs

The variant calling workflow is an instructive example not only because it comprises several recurring processing steps in next-generation sequencing but also because it is easy to scale. Additionally, large sequencing studies are publicly available and each processing stage provides opportunities to parallelize.

We have executed the variant calling workflow on a cluster of 8 workstation computers of which four have 120 cores, two have 32 cores, and another two have 16 cores amounting to 576 cores in sum. The cluster was connected via a GlusterFS installation [30] spanning four computers. The variant calling workflow processes 8 GiByte of compressed DNA samples and a Human reference genome amounting to a size of 3 GiByte.

Below, we show an example log, containing just one entry. The full execution log constituting the variant calling workflow contains about 12300 entries.


```
{
  "history": [
    {
      "app": {
        "app_id": "2d450c5fb4eca8f0f086b179b1827ba9bb6d8b95b0b1e93ea3e1c146",
        "arg_bind_lst": [
          {
            "arg_name": "bam",
            "value": "a0fb446_alignment.bam"
          }
        ],
        "lambda": {
          "lang": "Bash",
          "lambda_name": "samtools-sort",
          "ret_type_lst": [
            {
              "arg_name": "sorted",
              "is_list": false,
              "arg_type": "File"
            }
          ],
          "script": "\n  sorted=sorted.bam\n  samtools sort -m 2G $bam -o $sorted\n",
          "arg_type_lst": [
            {
              "arg_name": "bam",
              "is_list": false,
              "arg_type": "File"
            }
          ]
        }
      },
      "delta": {
        "app_id": "2d450c5fb4eca8f0f086b179b1827ba9bb6d8b95b0b1e93ea3e1c146",
        "result": {
          "ret_bind_lst": [
```

```
          {
            "arg_name": "sorted",
            "value": "3e1c146_sorted.bam"
          }
        ],
        "node": "cf_worker@gruenau6",
        "status": "ok",
        "stat": {
          "sched": {
            "t_start": "1548318431386045115",
            "duration": "1070343606"
          },
          "run": {
            "t_start": "1548318431452760423",
            "duration": "510892039"
          },
          "stage_in_lst": [
            {
              "size": "2820395",
              "filename": "a0fb446_alignment.bam",
              "t_start": "1548318431389949248",
              "duration": "62600273"
            }
          ],
          "stage_out_lst": [
            {
              "size": "2593264",
              "filename": "3e1c146_sorted.bam",
              "t_start": "1548318432088860324",
              "duration": "365315947"
            }
          ]
        }
      }
    }
  ]
}
```


6.2.1. Worker Allocation over Time

Executing the variant calling program with a 8 GiByte sample data set results in roughly 12300 function applications. Many of these function applications are independent and the execution environment can, thus, execute these in parallel. Furthermore, each function application requires its input data to be staged in from the distributed file system before execution starts and its output data staged out after execution has terminated successfully. Consequently, of the worker processes that are active, some workers stage data while others run function applications. To judge the configuration of the cluster

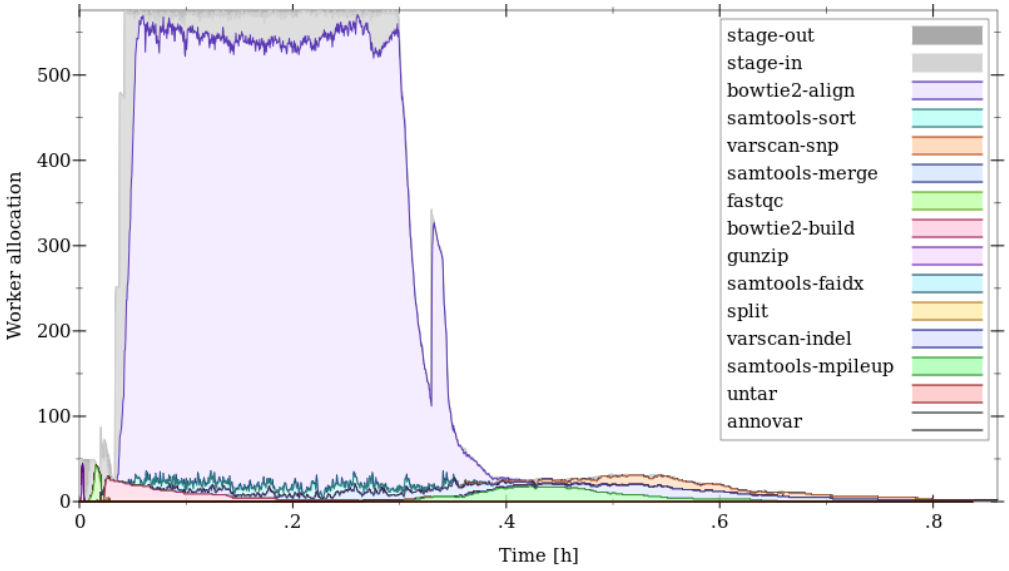


Figure 6.1.: Worker allocation over time for variant calling workflow executed in on a 8 node cluster amounting to 576 cores. Stage-in/out operations are shown in gray. The execution lasts 51 minutes; the cluster exhibits full load over a period of 18 minutes.

and the ability of the distributed runtime environment to supply function applications to its workers, it is important to know the worker's rate of execution to staging operations at each given point in time.

Figure 6.1 shows a worker allocation graph which stacks the amount of worker processes allocated for a specific operation on the ordinate and displays the time on the abscissa. Of the 51 minutes of execution time the cluster exhibited full load for about 18 minutes which is inherent in the structure of the workflow and the parallelism potential of the different processing stages. Also, the alignment function *bowtie2-align* consumes the bulk of the allocated resources.

6.2.2. Data Dependencies

A workflow's foreign function applications, while scheduled as independent chunks of work, have implicit data dependencies. Here we define a data dependency among foreign function applications, input-, and output data items as follows:

Definition 3. A data item is an input data item if and only if there is no application in the workflow that has produced it. An application depends on an input data item if and only if it consumes that data item.

Definition 4. A data item is an output data item if and only if there is no application that consumes that data item. An output data item depends on an application if and only if that application produces it.

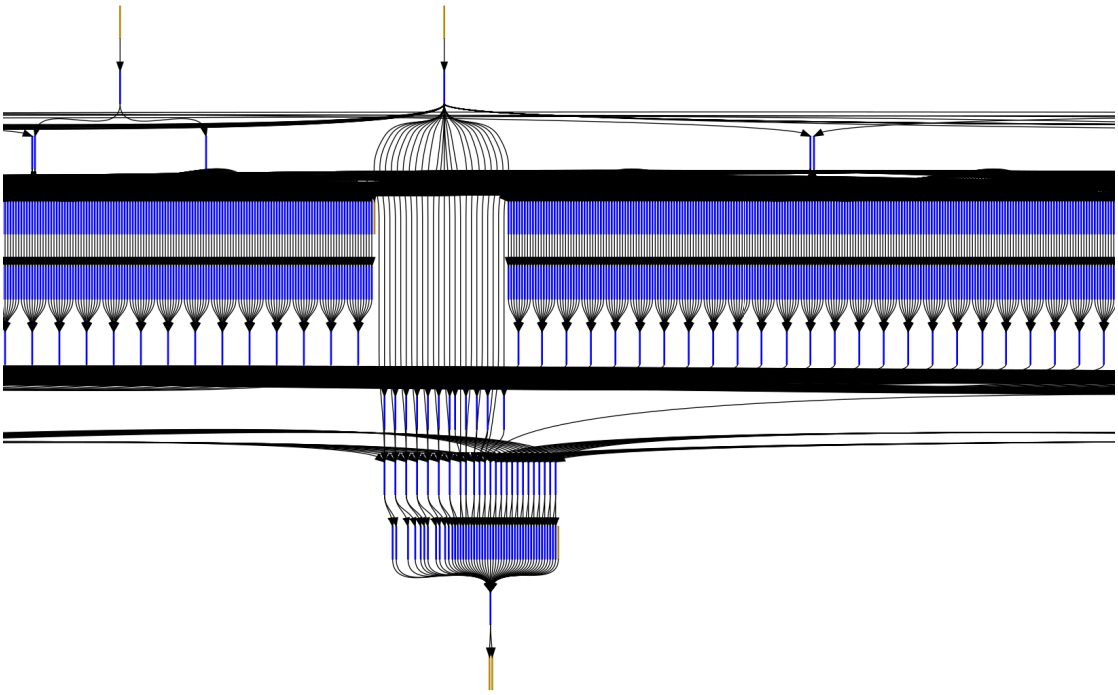


Figure 6.2.: Detail of the variant call dependency graph. Yellow vertical lines are input and output data. Blue vertical lines are foreign function applications. Black arrows are data dependencies.

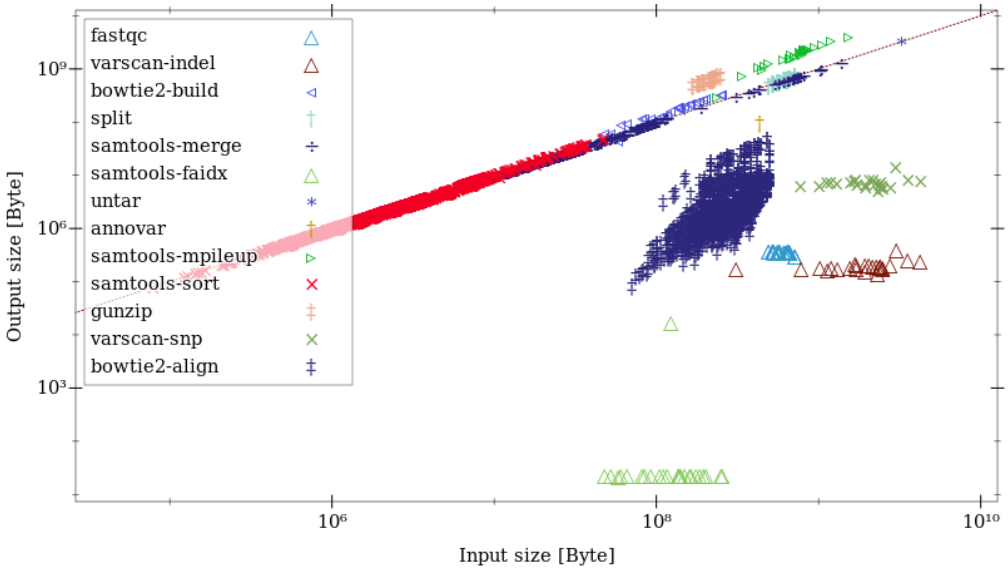


Figure 6.3.: Selectivity plot for variant calling workflow.

Definition 5. An application depends on an another application if the latter produces one or more data items that the former consumes directly.

Since the log contains for each foreign function application a list of which files it produced and consumed, we can construct a dependency graph from it. Note that we could not derive such a dependency graph statically from the workflow script alone. This is because the result of some of the operations Cuneiform provides can be determined in no other way than running the workflow.

Figure 6.2 shows a detail of the dependency graph that the variant calling workflow spans. The whole dependency graph is very wide, which would prevent us from distinguishing applications. Thus, we show only a detail here.

6.2.3. Function Selectivity

A function application takes an input data set and processes it to produce an output data set. Often, a function application's output is smaller than its input data set. Thereby, the processing step extracts the relevant information. We call the rate at which a function reduces the size of a data set its selectivity. This is analogous to the selectivity of database operations which describe the rate at which applying a database operator reduces the size of an input relation. Observing the selectivity of a function application allows us to anticipate the size of an output data set given the kind of function and the size of the workflow input. The output size estimate, in turn, allows us to improve scheduling decisions. Note that the current implementation of the distributed execution environment does not take the selectivity into account; we only observe it through collecting execution logs.

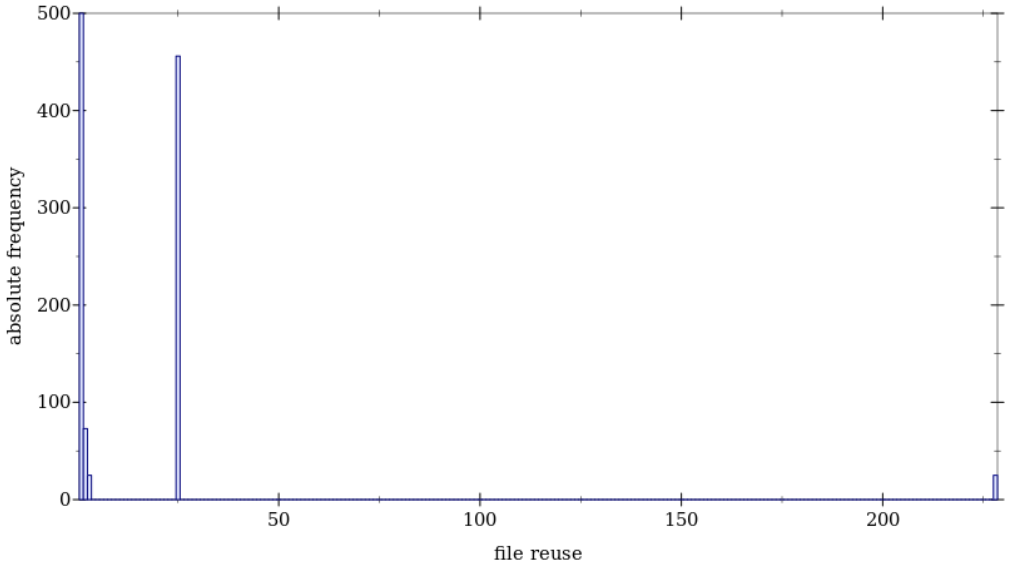


Figure 6.4.: File reuse histogram for variant calling workflow.

Figure 6.3 shows a selectivity plot for the variant calling workflow. The selectivity plot contrasts a function application's input data size and its output data size. Both axes are displayed on a logarithmic scale. Data points above the main diagonal inflate the amount of data circulating in the workflow, e.g. the *gunzip* function which unpacks a compressed file. Data points below the main diagonal deflate the data, e.g., *samtools-faidx* which creates an index much smaller than its input.

6.2.4. File reuse

The entirety of foreign function applications in a workflow uses some files only once while uses other files repeatedly. Knowing how large the portion of reused files is and how often these files are reused allows us insights into how we can lower staging costs by adjusting replication settings in the distributed file system and how much of a difference it makes to place applications on machines that locally store replicas of their input files.

Figure 6.4 shows a file reuse histogram. From the histogram we learn that the overwhelming majority of files are used only once. We clip the plot at an absolute frequency of 500 files. We also learn that about 450 files are reused 25 times. Finally, about 25 files are reused more than 250 times. From the structure of the variant calling workflow, we construe that the 25 files that are reused so often are the 25 reference genome indice.

6.2.5. Foreign Function Application Throughput

Every foreign function application that the distributed execution environment executes consumes input data and takes a certain time to complete. We observe both the size of the input data and the time to complete and log this information. We define the

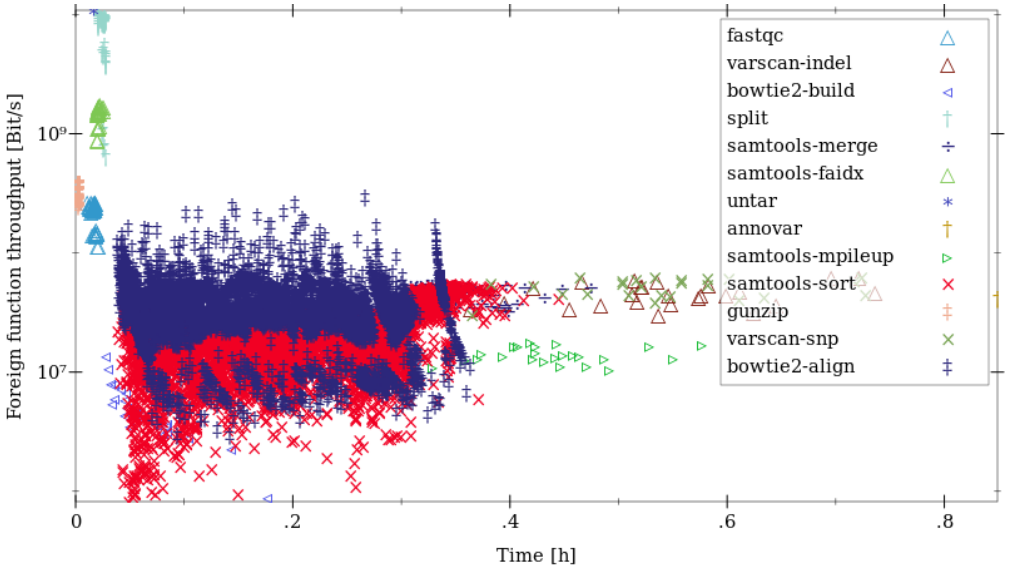


Figure 6.5.: Foreign function throughput over time for variant calling workflow.

throughput of a foreign function application as the quotient of the input data size and the time to complete. The throughput depends on various factors, e.g., the time-complexity of the software that makes up the foreign function, the performance of the computer that hosts the worker responsible for executing the application, or congestion of critical compute resources by operations running in parallel with the application.

Figure 6.5 visualizes the application throughput over time in a scatter plot. The abscissa displays the time from the start of the workflow run in hours. The ordinate displays the throughput of a single foreign function application in Bits per second on a logarithmic scale. Herein, we show each foreign function type in a different color.

Another way to visualize the foreign function application throughput is to estimate the probability density for a given throughput value. Figure 6.6 shows the throughput density for the variant calling workflow. Some foreign function applications have a very high throughput. Thus, we clip the plot at 0.5 GiBit per second. The majority of applications exhibit a throughput of 0.025 GiBit per second. Foreign function applications with a low throughput are CPU-bound, while applications with a high throughput are I/O-bound.

The throughput density may depend on many factors. For example, the throughput density may depend on the machine that hosts the worker process: we expect a fast machine to have a higher throughput than a slow machine. Figure 6.7 shows the throughput density dependent on the machine that executes a foreign function application. All eight workstation computers have a similar application throughput of around 0.025 GiBit per second. Thus, we cannot, by close observation, discern a large difference between any of the eight machines. The situation might be different in a more heterogeneous cluster though.

Another variable that might influence the throughput is the type of foreign function

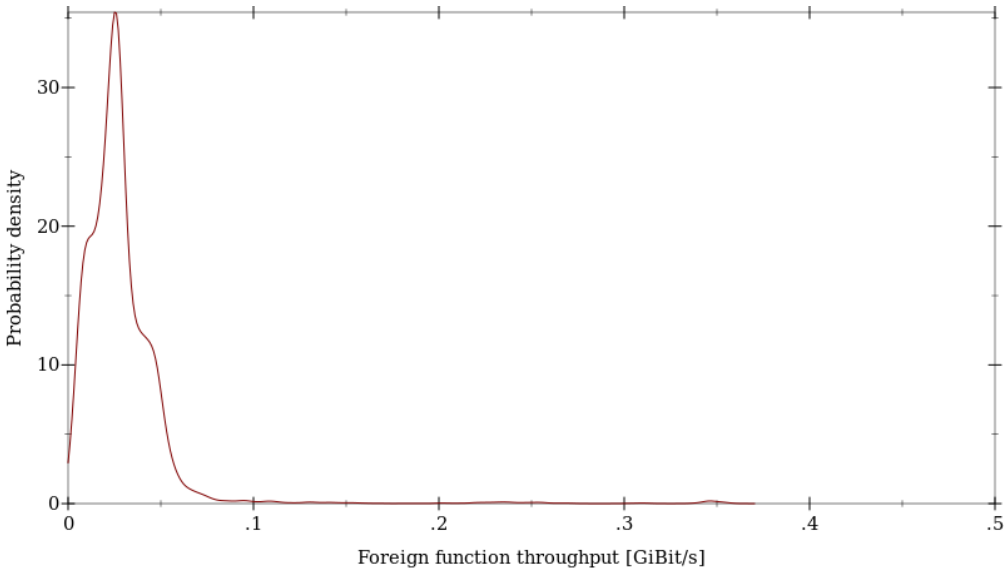


Figure 6.6.: Foreign function throughput density for variant calling workflow.

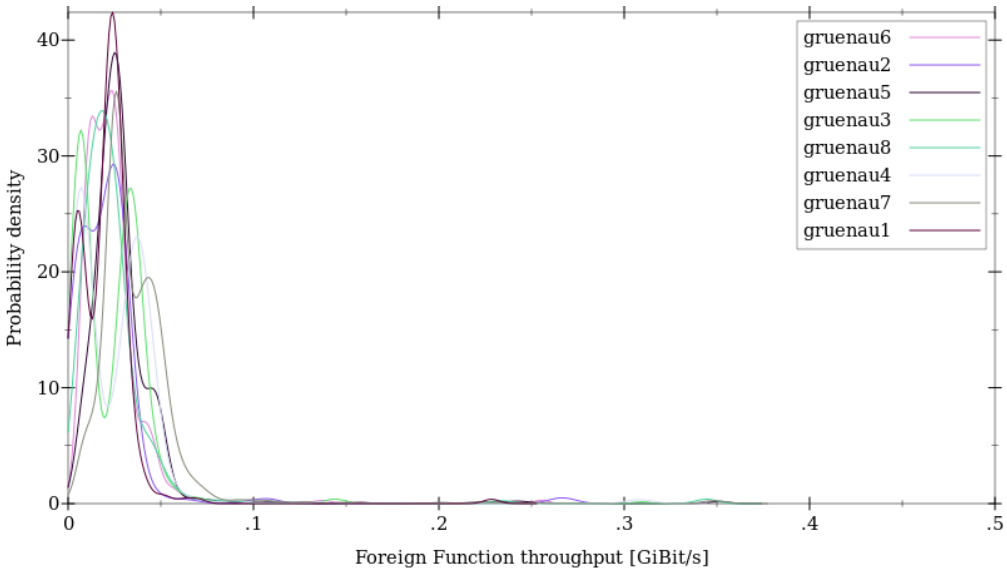


Figure 6.7.: Foreign function throughput density for each host for variant calling workflow.

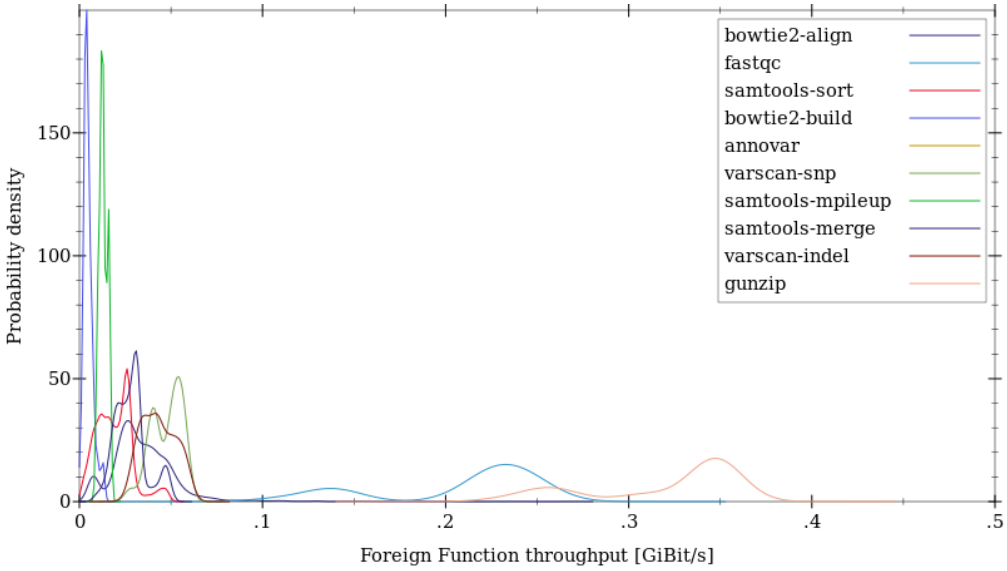


Figure 6.8.: Foreign function throughput density for each function type for variant calling workflow.

in a foreign function application. Some foreign functions might, overall, be I/O-bound while others might be CPU-bound. Figure 6.8 shows the throughput density dependent on the function type of the application. We observe that the functions *bowtie2-build* and *bowtie2-align* have a particularly low throughput. Furthermore, *samtools-sort*, *samtools-mpileup*, *samtools-merge*, *varscan-snp*, and *varscan-indel* have somewhat larger throughput. All other function types are I/O-bound. Thus, by close observation, we find that the throughput does depend on the function type of a foreign function application.

6.2.6. Staging Bandwidth

Data transfers impact the execution time of a workflow. The higher the bandwidth between the host to execute a foreign function application and the distributed file system and the smaller the data to transfer, the smaller is the impact of data transfers on the workflow execution time. Conversely, in large-scale data analysis, the bandwidth of data transfer becomes a dominating factor. Thus, we need to measure the time that the distributed execution environment is using for data transfers. Figure 6.1 in Section 6.2.1 already gave us an impression about how much time the distributed execution environment spends for data transfers.

To visualize data transfers we plot the data transfer time for each stage-in and stage-out operation performed during workflow execution. Figure 6.9 shows a scatter plot for all staging operations that occurred. Although the workstation computers are connected via a 1 GiBit per second switch, some staging operations perform much faster than that. The reason is that the operating system caches file system accesses. However, the majority of data transfers is below the 1 GiBit per second mark.

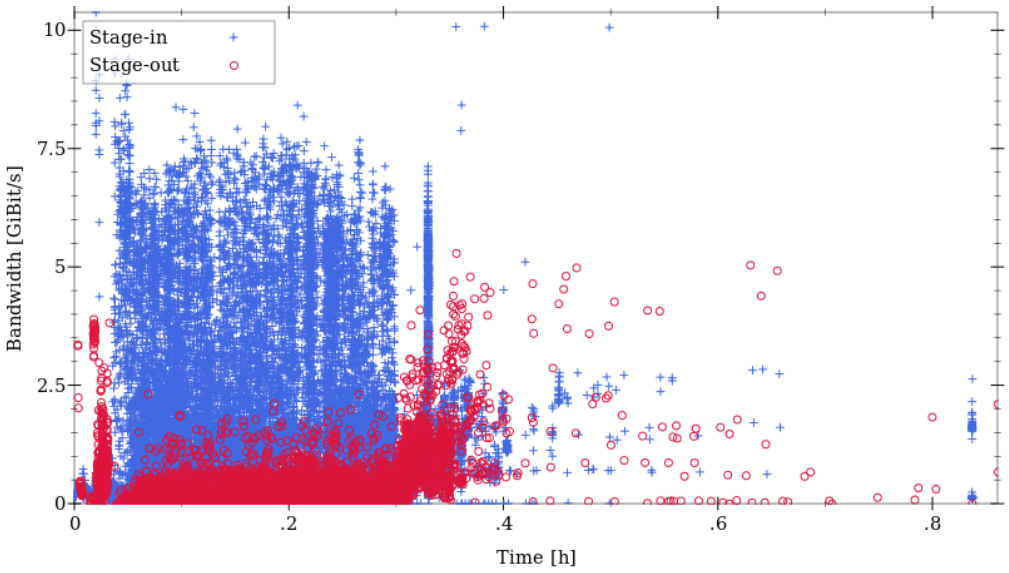


Figure 6.9.: Staging bandwidth over time. Blue crosses are stage-in operations. Red circles are stage-out operations.

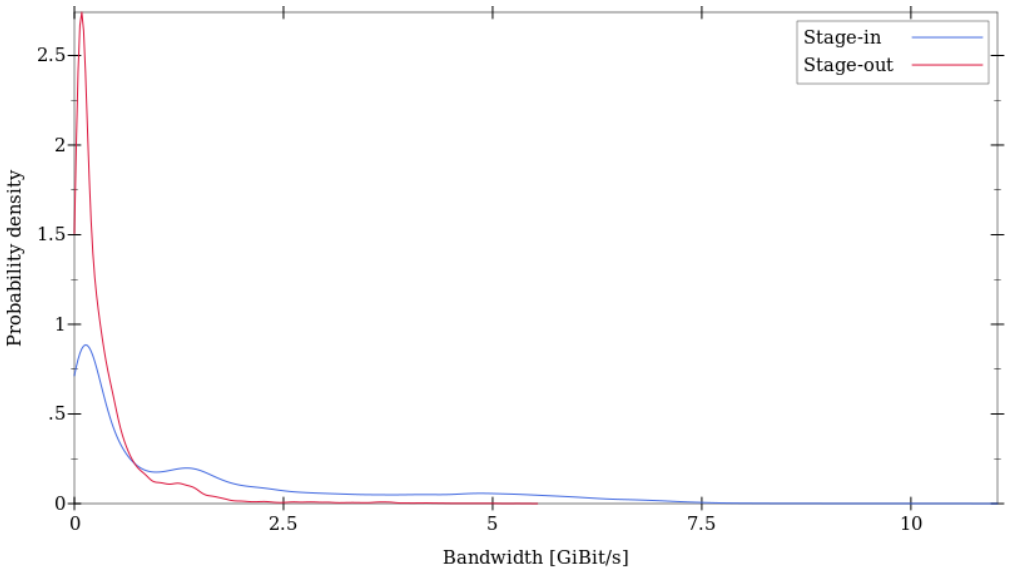


Figure 6.10.: Staging bandwidth density.

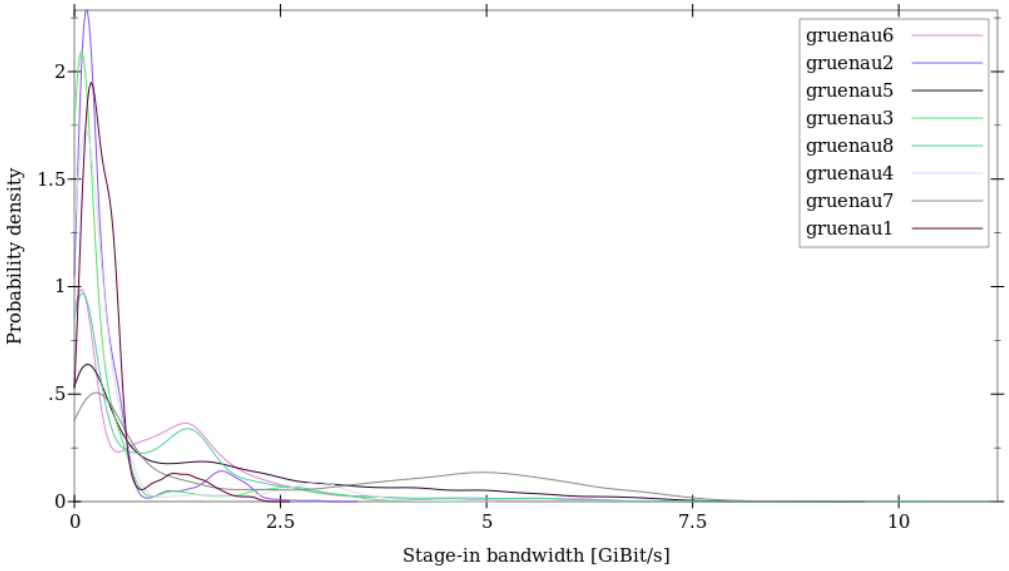


Figure 6.11.: Stage-in bandwidth density for each machine.

In Figure 6.10 we plot the probability density for a data transfer operation having a particular bandwidth. The density function shows more clearly than the scatter plot that only a small fraction of data transfers are served by cache hits or local writes. It also confirms the tendency of stage-in operations taking less time in tendency than stage-out operations.

The speed of stage-in and stage-out operations may vary across various computers. Some machines may have better network adapters and the connection topology might favor some machines while neglecting others. To find out the differences between the machine's stage-in and stage-out bandwidths we plot the bandwidth density individually for each machine. Figure 6.11 shows the stage-in bandwidth density for each machine. Similarly, Figure 6.12 shows the same plot for the stage-out bandwidth. By close observation we find that the differences in the network bandwidth among the workstation computers is small.

6.3. RNA-seq

In this section we discuss an RNA-seq workflow. The RNA-seq application is especially useful to show how libraries with different language interfaces can be integrated into Cuneiform. Concretely, we perform a large portion of the analysis using command line tools. However, we use an R library to carry out the visualization. With Cuneiform, we wrap both command line scripts and R scripts into foreign functions with a uniform interface. Calling these foreign functions forms the RNA-seq workflow. We analyze sequenced transcription data of moderate size. Like the DNA-seq workflow we discussed in the previous section, this workflow consists of several subsequent steps entailing trim-

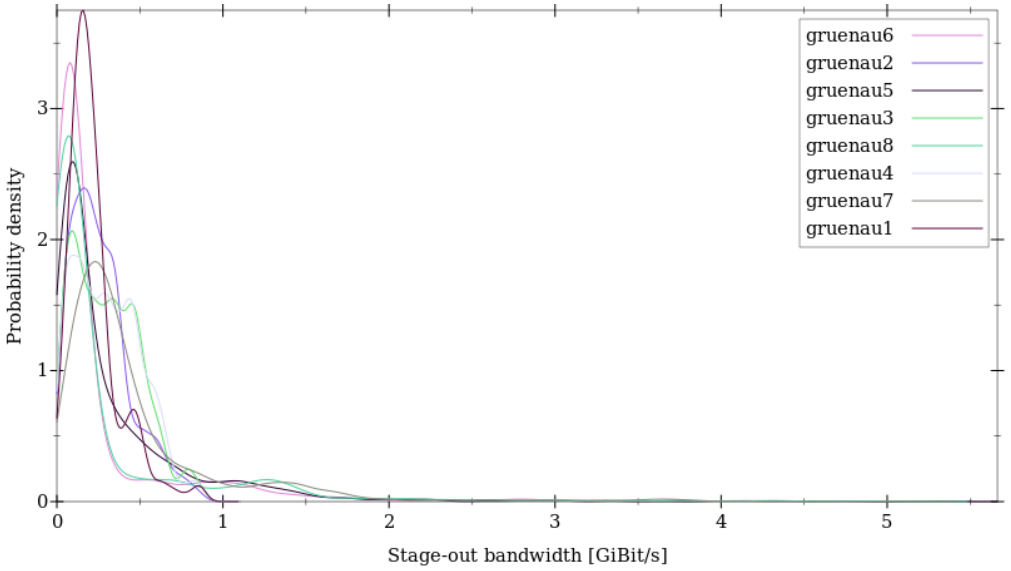


Figure 6.12.: Stage-out bandwidth density for each machine.

ming, alignment, junction mapping, and visualization.

We use RNA-seq to explore an organism's transcriptome. The transcriptome entails all genes a cell transcribes from DNA to RNA. Detecting if RNA is present or measuring how much RNA a cell transcribes in a given condition allows us insight in the effects of gene regulation. Using RNA-seq can help discover new genes or splice variants of known genes. Also it allows to assess the difference in gene expression under different conditions [164, 237]. We give the Cuneiform workflow script in Appendix C.2

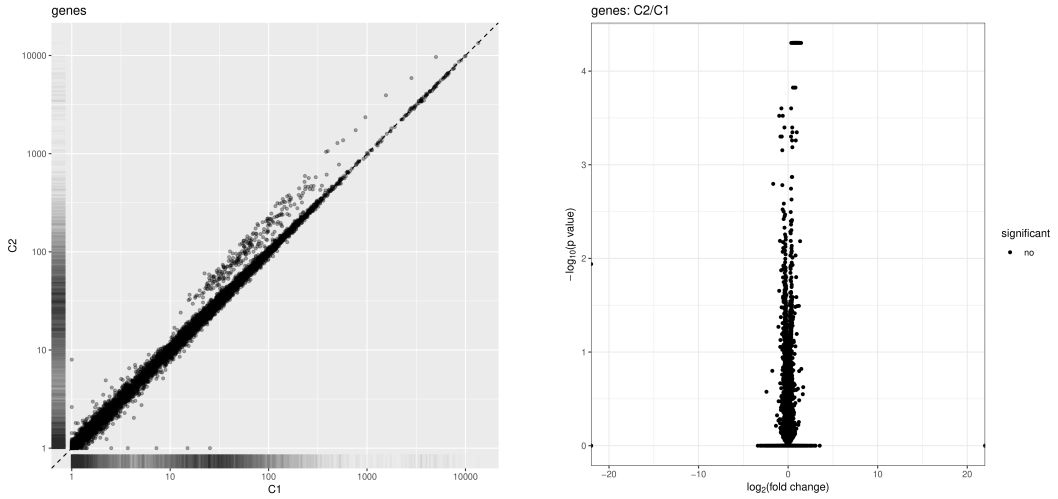
6.3.1. Methods

Here, we recreate a protocol by Trapnell et al. [222]. The paper shows a method of finding differences in gene expression using RNA-seq. The first step of an RNA-seq protocol is to extract RNA material from an organism. We back-transcribe the RNA material to complementary DNA material, which we, then, sequence using DNA sequencing.

This particular study uses simulated transciptomic data that we obtained from the gene expression omnibus [15, 67]. The data set simulates transcripts from the fruit fly *Drosophila melanogaster*. We obtain a reference genome and gene index for this organism from the Illumina iGenomes website[2].

In their 2012 paper Trapnell et al. use TopHat [221], a fast splice junction mapper to analyze RNA data. They use Cufflinks[3] to assemble the mapped transcripts and estimate their abundances. In addition, they use Cufflinks to compare the two test conditions and identify differentially expressed genes. Lastly, Trapnell et al. use the R library CummeRbund [88] to visualize their results.

[2] `https://support.illumina.com/sequencing/sequencing_software/igenome.html`
[3] `http://cole-trapnell-lab.github.io/cufflinks/`



(a) Scatter plot comparing the gene transcription levels of conditions C1 and C2 on a logarithmic scale

(b) Volcano plot showing the log-ratio between the gene transcription levels of the conditions C2 versus C1

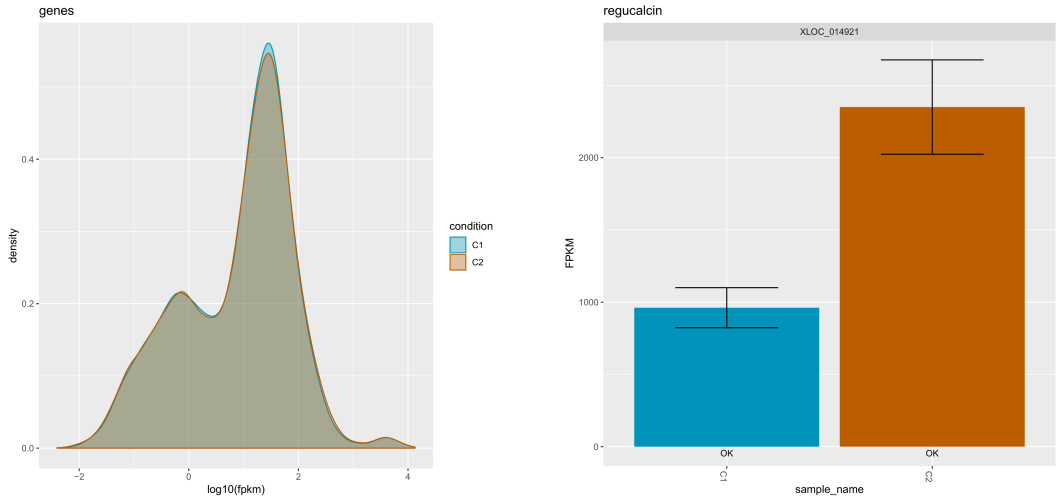
Figure 6.13.: Scatter plots comparing overall gene expression levels between conditions C1 and C2

At this time, the TopHat project has been superseded by HISAT2 [130] and has gone into a low maintenance mode. As a consequence, it has become incompatible with current versions of GCC. Thus, as the study ages it becomes increasing challenging to recreate all of the involved software. Here, instead of transposing the study to a contemporary setting, we reproduce every step as is sticking with TopHat that was last updated in 2016. As a consequence, we also have to downgrade Bowtie to a 2016 version. Both TopHat and Bowtie in their 2016 versions are incompatible with recent releases of the C compiler in the open-source compiler collection GCC. Thus, we have to downgrade GCC to version 5.5.0 to compile both tools.

6.3.2. Results

We compare the gene transcription levels with Cufflinks and visualize the differences between the conditions C1 and C2 using the CummeRbund library. When visualizing transciption level differences we can either compare all transcribed genes or we can compare the differences between one particular gene. We can even distinguish various isoforms of a gene, further narrowing the scope of the comparison.

Figure 6.13a shows a scatter plot in which each data point represents a gene. Herein, the x axis shows the logarithmic transcription levels for condition C1 and the y axis shows the logarithmic transciption levels for condition C2. When a data point lies above the main diagonal, this gene is up-regulated in C2 compared to C1; when it lies below, this gene is down-regulated. We can see that the majority of genes remains close to the main diagonal while some genes show distinct up-regulation.



(a) Comparison of the probability densities of logarithmic fragment count per kilobase million for conditions C1 and C2

(b) Comparison of the regucalcin fragment count per kilobase million for conditions C1 and C2

Figure 6.14.: Fragment count per kilobase million comparing conditions

Figure 6.13b shows a volcano plot that visualizes the gene transciption change between the two conditions and the confidence we ascribe to this change. Each data point represents a gene. Herein, the x axis shows the logarithmic difference between the transcription levels of condition C1 and C2. On the y axis plots show the logarithmic p-value of the observation. The higher up a data point appears, the more confident we are about its transcription level change.

The scatter plots give a first visual impression of the transcription level changes between conditions. Differences are easy to spot in these plots although these differences are small overall. Figure 6.14a shows the estimated probability density function for the logarithmic transcription levels for both conditions C1 and C2. Both densities are almost identical.

Only a small number of genes actually differ in their transcription levels. One of these genes is the regucalcin gene. Figure 6.14b shows the regucalcin transcription levels in both conditions C1 and C2. In condition C2 the regucalcin transcription level is elevated by a factor of about two.

One gene can be transcribed into several isoforms. A gene's isoforms are different messenger RNAs that come from the same location in the genome. The regucalcin gene shows four different isoforms each with different transcription levels. Figure 6.15 shows the transcription levels of the four regucalcin isoforms in both conditions C1 and C2. The plot shows that only the second regucalcin isoform has actually changed its transcription level by a factor of three while all other isoforms remain unaffected.

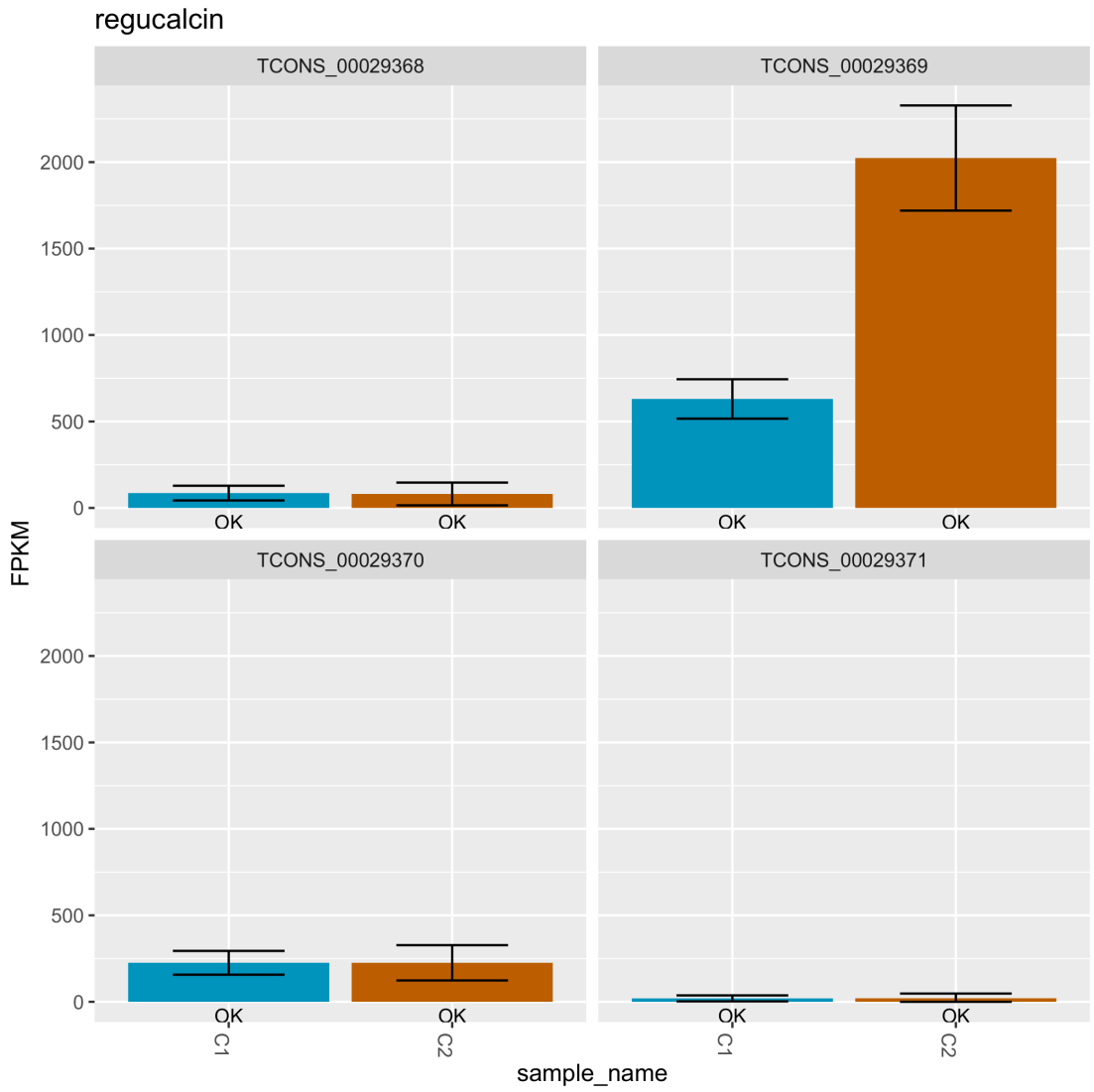


Figure 6.15.: Comparison of fragment count per kilobase million for conditions C1 and C2 contrasting four isoforms of the regucalcin gene

6.3.3. Discussion

The RNA-seq workflow shows how Cuneiform integrates tools with different interfaces: TopHat and Cufflinks are commandline tools which we drive in Bash while CummeRbund is an R library. However, the parallelization potential of this workflow is limited. Under the hood, TopHat and Cufflinks are workflow systems in their own right. TopHat manages Bowtie, SAMtools, and a number of built-in scripts. Cufflinks too hides index building, processing, and data transformations behind a commandline interface. We can parameterize both tools to use several cores to perform independent processing steps.

However, for Cuneiform both TopHat and Cufflinks are black boxes. This means that Cuneiform can neither distribute independent intermediate steps nor can it reuse intermediate results for the parts of the workflow that are managed by TopHat and Cufflinks. The only remaining opportunity for parallelization is the independence between the analysis of the two conditions. This means that for this RNA-seq workflow Cuneiform can saturate a cluster larger than two workstations only in the alignment phase of the workflow.

6.4. ChIP-seq

We use ChIP-seq to analyze protein interactions with DNA.[4] ChIP-seq consists of two steps: First, we select DNA fragments using a ChIP essay and sequence the selected DNA fragments. Second, we match all sequenced DNA fragments to a known reference genome and count the matching DNA fragments in each location. This allows us to find the DNA sites a protein binds to. Analyzing protein interactions with ChIP-seq allows us to study gene regulation and epigenetic mechanisms to better understand diseases and biological pathways [178].

Here, we reproduce the following workflow from a study by Myers et al. 2013 [167]. The study explores gene expression in bacteria under anaerobic conditions in *Escherichia coli*. We first perform a genome-wide ChIP-seq analysis. For the later stages of the workflow we focus on the FNR binding sites in the bacterial genome. We give the Cuneiform workflow script in Appendix C.3.

6.4.1. Methods

The input data to the ChIP-seq analysis consists of two sequence data sets: The first data set contains DNA samples gathered from *Escherichia coli* without selecting specific DNA sites. The other DNA sample contains only DNA fragments that have been selected by a FLAG-tagged FNR protein in a ChIP protocol. In addition, we need an *Escherichia coli* reference genome.

First, we perform quality control over the sequence data sets using FastQC[5]. This way we convince ourselves that the sequence data does not contain adapters or leading or trailing low quality bases. Next, we use Bowtie [139] to align both the tagged sample and

[4] `https://github.com/joergen7/chip-seq`
[5] `https://www.bioinformatics.babraham.ac.uk/projects/fastqc/`

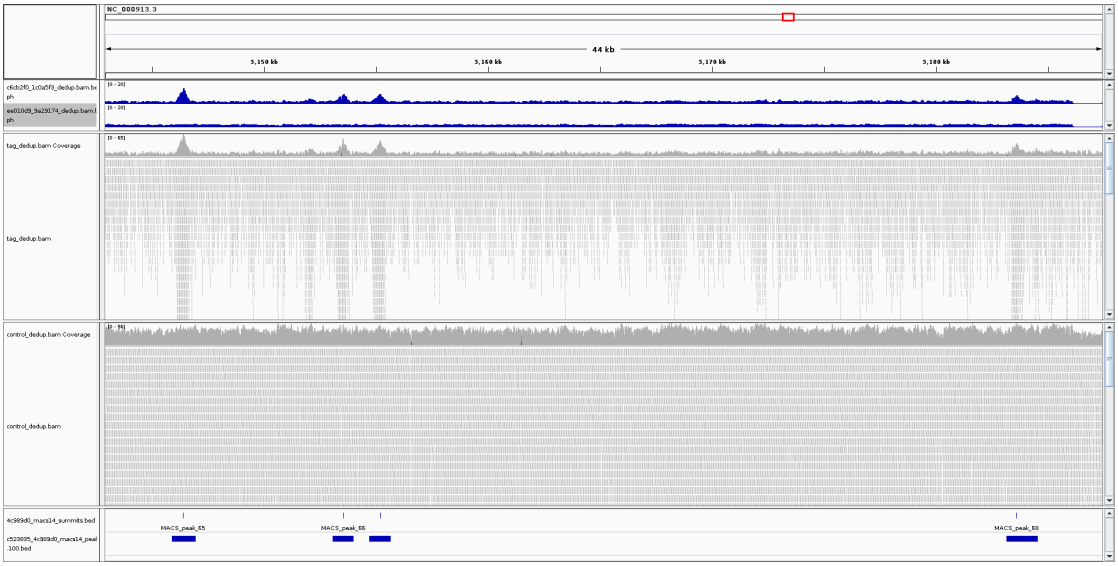


Figure 6.16.: ChIP-seq called peaks viewed in IGV. The top lanes show normalized coverage for both tagged sample and control group extracted with deepTools. The middle lanes show the sequence alignments to the reference genome for both tagged sample and control group. The bottom lanes annotate peak regions and summits called by MACS.

the control group to the *Escherichia coli* reference genome. After aligning both samples to the reference genome, some regions have a higher coverage than others. We use MACS to compare the coverage in the tagged sample with the coverage in the control group to find statistically significant peaks in the coverage of the tagged sample. This step is called peak calling. To further process the DNA sequence of the called peaks we restrict each peak to an area of 100 base pairs to the left and to the right of its center using a custom Perl script. Lastly, we use bedtools [189, 190] to extract the DNA sequence of the called peaks. The output of applying MACS is a definition of peak regions and summits in the form of an annotation file in bed format.

In addition to peak calling we use deepTools [191] to visualize the coverage information for both the tagged sample and the control group. To do that we need an indexed and duplicate-free version of the alignments which we obtain using SAMtools [143]. We apply deepTools to the processed alignment normalizing coverage by the reads per genomic content (RPGC). Herein, the genomic content is the effective length of the *Escherichia coli* genome of 4.6 million base pairs.

We visualize the normalized coverage information, the peak annotations called with MACS along with the alignments generated with Bowtie using the integrative genome viewer IGV [197, 218] (see Figure 6.16).

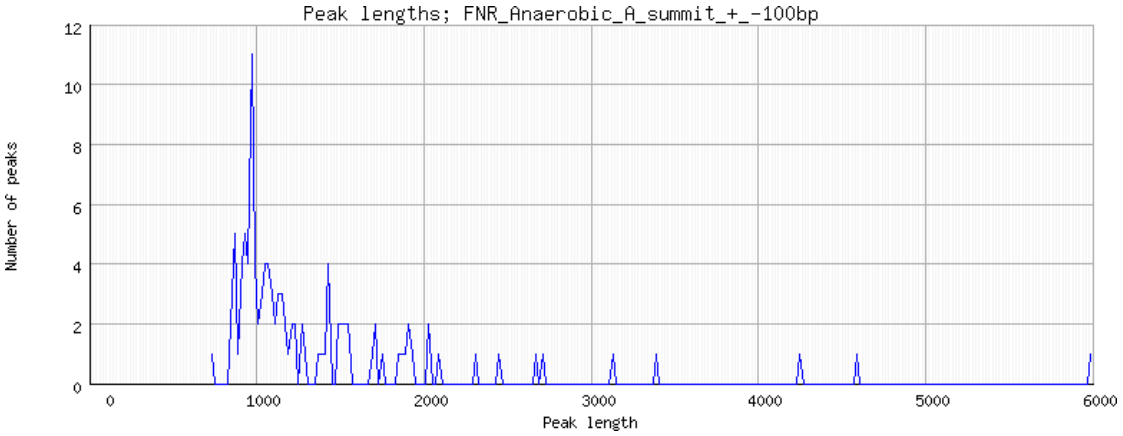


Figure 6.17.: Peak length distribution. Most peaks have a length of about 1000 base pairs.

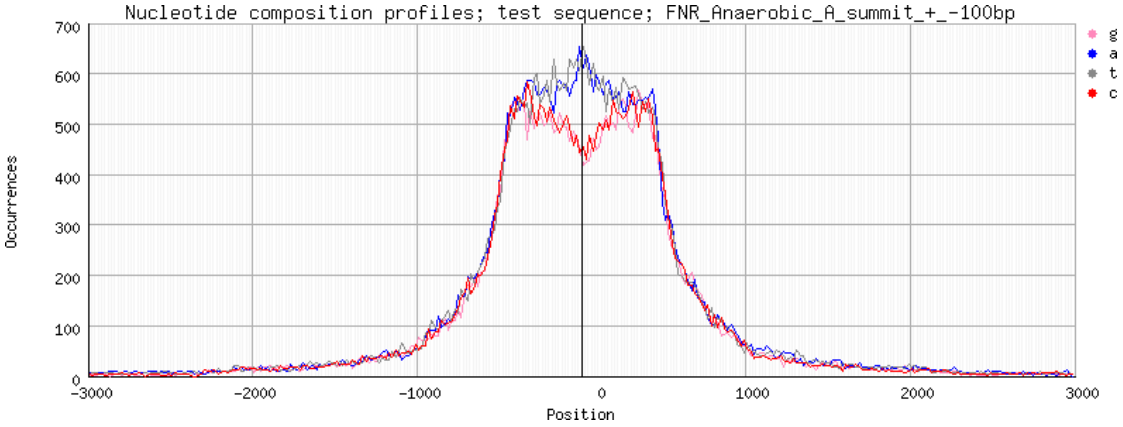


Figure 6.18.: Peak nucleotide distribution relative to summit position

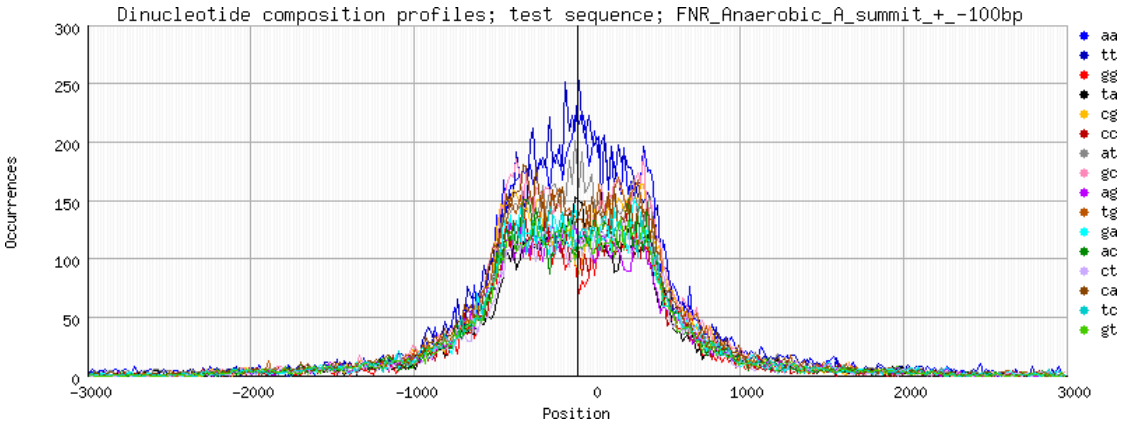
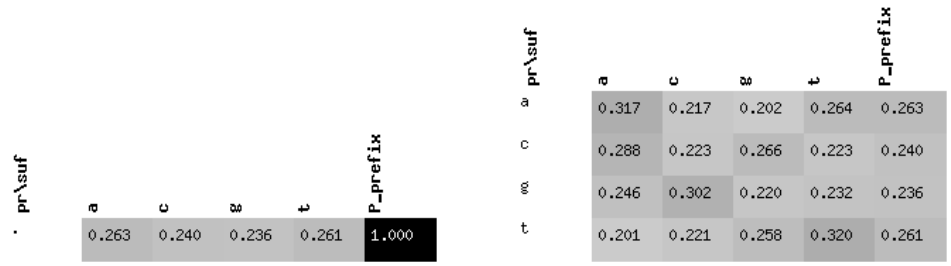
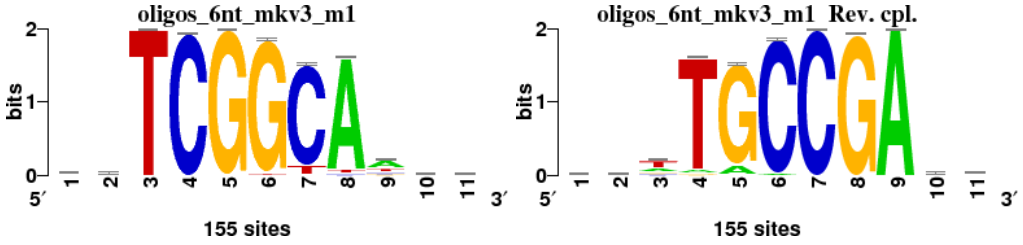


Figure 6.19.: Peak nucleotide digram distribution relative to summit position



(a) Peak nucleotide probabilities (b) Peak nucleotide digram probabilities

Figure 6.20.: Peak nucleotide probabilities heatmap



(a) Forward strand motif (b) Reverse strand motif

Figure 6.21.: Common DNA oligonucleotide motif discovered using RSAT

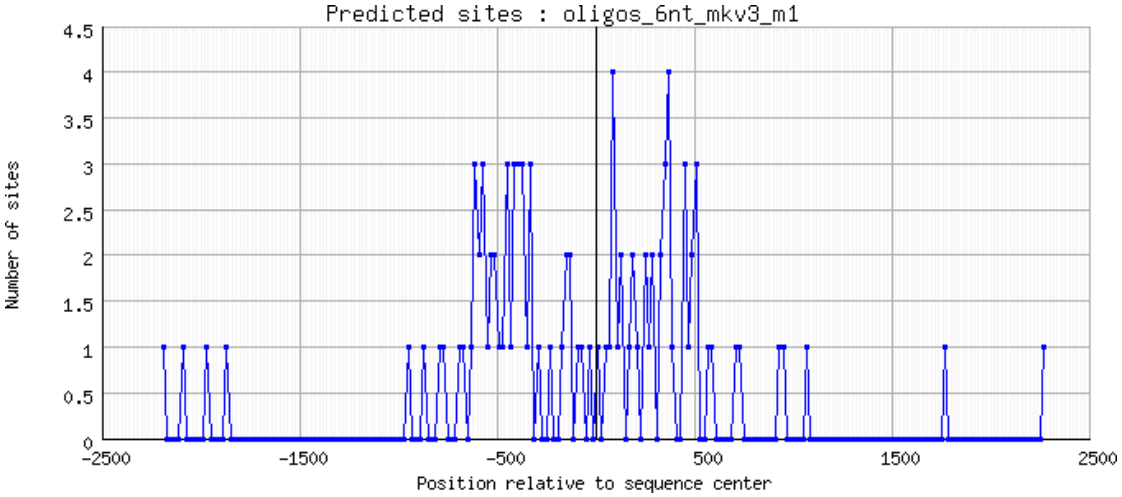


Figure 6.22.: Position distribution of a common motif relative to the peak center

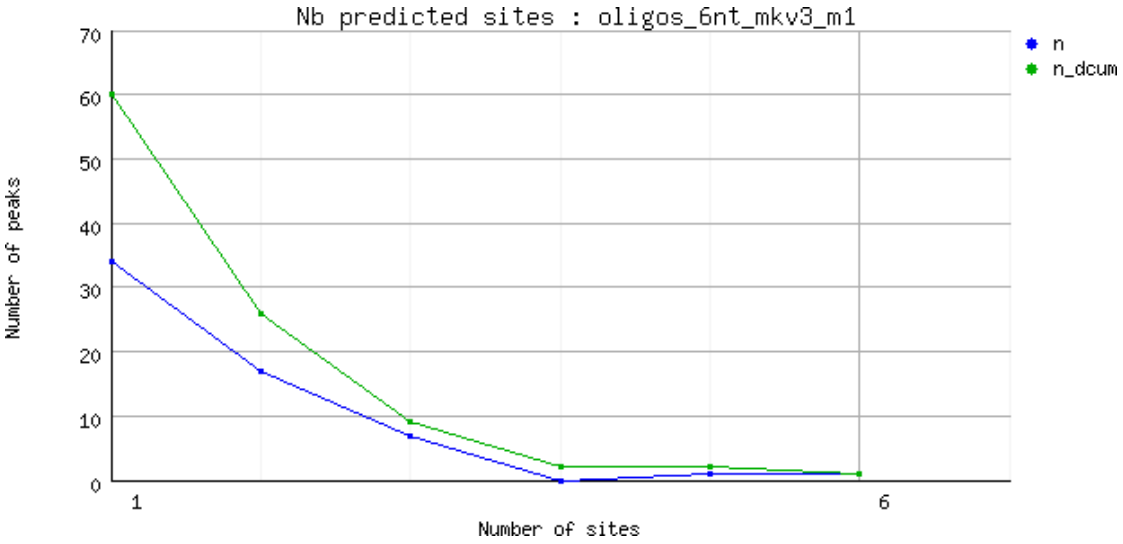


Figure 6.23.: Number of peaks with at least n predicted copies of a motif. The motif appears up to six times inside a peak.

6.4.2. Results

We have shown how MACS can be used to call peaks from ChIP-seq data sets. Such peaks are usually further processed for visualization or motif discovery. Herein, a gene motif is a short DNA sequence characterizing a peak. E.g., if many peaks contain one particular gene motif, that means that the motif probably plays a part in the DNA-binding process we study. We use the tool RSAT [106, 216] to visualize both statistical properties of the called peaks and discovered motifs. Figure 6.17 shows a histogram visualizing the peak length distribution. The nucleotide and nucleotide digram distributions around the peak center are visualized in Figures 6.18 and 6.19. The heatmaps shown in Figure 6.20 show the nucleotide probabilities inside the peak regions.

We can further use RSAT to find motif sequences inside the peak call data. Figure 6.21 shows the HMM profile of the most prolific motif found in the peak call data set. Around the peak regions this motif appears 155 times. Figure 6.22 shows a distribution of the position of the motif relative to the peak center. Lastly, the motif may appear several times inside a peak region. Figure 6.23 shows the number of peaks holding at least n copies of the motif for up to six copies.

6.4.3. Discussion

There are two opportunities for Cuneiform to increase parallelism: First, the tag and control samples are independent up to the peak calling step, i.e., quality control, read mapping, and alignment post processing can be parallelized. Second, peak calling with MACS and extracting the coverage information with deepTools are also independent. In all, the ChIP-seq workflow provides only limited opportunity for parallelization.

6.5. Phylogeny Analysis

In this section we show a phylogeny workflow. The phylogeny workflow is especially suitable to show how Cuneiform integrates the programming language Awk [4, 3] to process FastA sequence annotations. The workflow processes amino acid sequences from many organisms. These amino acid sequences undergo preprocessing, homolog search, multiple sequence alignment, and hierarchical clustering.

A phylogenetic tree shows the evolutionary relationship between several species by grouping together similar species while keeping apart different species. Such an analysis often compares a specific aspect the species share. Here, we study the phylogeny of a protein domain: The CHASE domain [12]. This protein domain belongs to a signalling system known as the two-component system [153]. We can find the CHASE domain not only in many bacteria but also in plants. This suggests that the CHASE domain has been conserved throughout the evolution from bacteria to land-inhabiting plants. If this evoluationary relationship between bacteria and plants exists, we should find the CHASE domain also in algae which stand between bacteria and plants. We locate the CHASE domain in the proteomes of plants, bacteria, and algae and study its phylogeny. Thereby, we loosely repeat a phylogenetic analysis performed by Heyl et al. 2007 [109] and refined by Pils and Heyl 2009 [184].

The two-component system is a signal transduction mechanism. As the name suggests it consists of two components: (i) the receptor protein, a Histidine kinase [239], that sits at the cell membrane and reacts to a signal received on the outside by changing its conformation and (ii) a regulator protein that detects the conformation change of the receptor protein on the inside and relays the so-received signal to the cell's nucleus where it regulates genetic expression.

The two-component system's Histidine kinase protein consists of several protein domains. Here, we are interested in the protein domain that is responsible for receiving the hormone signal from outside the cell. The CHASE domain is such a receptor domain. In plants, the CHASE domain is sensitive to cytokinins such as kinetin, zeatin, or 6-benzylaminopurine.

We find two-component systems in a large variety of bacteria and plants. Also the CHASE domain is present not only in plants but in bacteria. However, for many of the bacterial CHASE domains it is unclear what hormone they are sensitive to. We give the Cuneiform workflow script in Appendix C.4

6.5.1. Methods

If the CHASE domain is present in bacteria and land-inhabiting plants it should be possible to find it in algae too, since algae and plants share common ancestors. To study the evolution of the CHASE domain we obtain a profile of the CHASE domain. Next we search for matches to this profile in the proteomes of a variety of species including bacteria, plants, and algae. From the amino acid sequences that match the CHASE domain profile we construct a multiple sequence alignment. Lastly, we use this multiple sequence alignment to construct a phylogenetic tree.

First, we need a definition of the CHASE domain in the form of a profile-Hidden Markov Model [63, 64]. We can obtain such a profile from the Pfam database [17]. We download a seed alignment of the CHASE domain which we use to find the CHASE domain in the proteomes of other species. Next, we need to obtain proteomes of the species of interest. Here we have used the amino acid sequences in the gene indices provided by Gramene[6] and the Joint Genome Institute[7]. Both the profile-Hidden Markov Model and the amino acid sequences form the input data for the phylogenetic analysis. In total, we search 583 gene indices from different species for the CHASE domain. The gene indices comprise 58 plant indices from Gramene, 26 gene indices from the JGI, including eight fungi that we use to test how susceptible the search algorithm is to false positives, and 499 microbial genomes from the JGI.

We perform the search for the CHASE domain using HMMER [16], a biosequence analysis tool using profile-Hidden Markov Models. We index the profile-Hidden Markov Model which comes in Stockholm format using HMMER. The amino acid sequences obtained from the gene indices in FastA format can be used as is, however the bacterial sequences obtained from the Joint Genome Institute come as genomic information. We translate these DNA sequences into amino acid sequences by finding all open reading frames in both the forward and the backward strand of the bacterial genome. For this we use the Easel tool suite that ships with HMMER.

The result of the HMMER search is a collection of amino acid sequences in FastA format which contain the protein domains that match the CHASE domain. We concatenate all HMMER results into one large FastA file and reformat the comment tags of each sequence to be consistent and human readable.

Like Pils and Heyl 2009 [184] we use MUSCLE [65, 66] for creating the multiple sequence alignment. MUSCLE can deal with very large alignments by iteratively improving a given alignment until it either finds a local optimimum or runs out of time.

Next, we create the phylogenetic tree. Heyl et al. 2008 use ClustalW [217] to obtain a phylogenetic tree using a neighbor joining clustering algorithm [169]. Neighbor joining is a fast but superficial method to derive a phylogenetic tree. As a greedy method the quality of the result degrades with large trees. Therefore, Pils and Heyl 2009 [184] use MrBayes [115] to perform the phylogenetic analysis. Here we use PhyML [96, 97, 98] a maximum likelihood-based method.

PhyML iteratively improves a phylogenetic tree. The initial tree is obtained using neighbor joining and is then improved by using subtree pruning and regrafting (SPR). We use SPR because the default algorithm using nearest neighbor interchanges (NNI) typically gets trapped in a local optimum showing occasional long branches in a group otherwise closely related siblings. As an amino acid substitution model we use the LG model proposed by Le and Gascuel 2008 [79].

To test our approach we have included the gene indice of several fungi and animals in our study. As we expected no CHASE domain hit was produced in any of these species.

In the last step we convert the phylogenetic tree that PhyML outputs in Newick format

[6] `http://www.gramene.org/`
[7] `https://jgi.doe.gov/`

to a graphics file using FigTree[8].

6.5.2. Results

Figure 6.24 shows the phylogenetic tree of the CHASE domain. It shows that most land-inhabiting plants are closely related. An exception are the moss *Physcomitrella patens* and the lycophyte *Selaginella moellendorffii* which show a much looser relationship both with other land inhabiting plants and also among their variants.

Like Pils and Heyl 2009 [184] we find CHASE domains in different algae like the colonial green alga *Volvox carteri*, the single-cell green alga *Chlamydomonas reinhardtii*. The diatoms *Phaeodactylum tricornutum* and *Thalassiosira pseudonana* we present have not been included in the 2009 study. In contrast, the CHASE domain reported in *Ostreococcus tauri* does not show up in our study because an E-value of 0.16 would be needed in the HMMER run to lift this hit above the detection threshold. This means that we would have to expect 16% of all hits to be false positives. For this study we used an E-value of 10^{-3}. This means that we expect one false positive in 1000 hits. Since we have found several hundred CHASE domains among plants and bacteria, we can be sure that no more than one of these hits is a false positive.

6.5.3. Discussion

Heyl et al. 2007 have reported the CHASE domain in a number of bacteria like *Pseudomonas syringae* or *Rhodospirillium ruburm*. We could reproduce this result. Furthermore, we can observe that the bacterial CHASE domains are widely different both between bacterial strands and among the CHASE domains found in a single strand.

We have included a gene index of the single-cell eukaryotes *Naegleria gruberi* and *Dictyostelium purpureum* which have produced CHASE domain hits. However, an interpretation of this finding exceeds the scope of this thesis.

6.6. Distributed *K*-means Clustering

This section shows a K-means workflow. This workflow is especially suited to show how we can use Cuneiform's recursion feature to repeat an iterative gradient descent algorithm until a convergence criterion is met. The workflow applies an expectation maximization scheme repeatedly that consists of two steps: (i) it associates points to a cluster and (ii) it reestimates the cluster center. The algorithm halts when the cluster-association does not change over two subsequent steps.

The K-means algorithm [148] is a popular algorithm for unsupervised labeling of a multidimensional data set. Herein, clustering partitions a data set into groups such that the observations in each group minimize a cost function. So the goal of a clustering algorithm is to find an optimal encoder function associating a data point to a group. The K-means algorithm describes how to find such an optimal encoder function by starting with an initial encoder and improving it in an iterative gradient descent [103].

[8] `http://tree.bio.ed.ac.uk/software/figtree/`

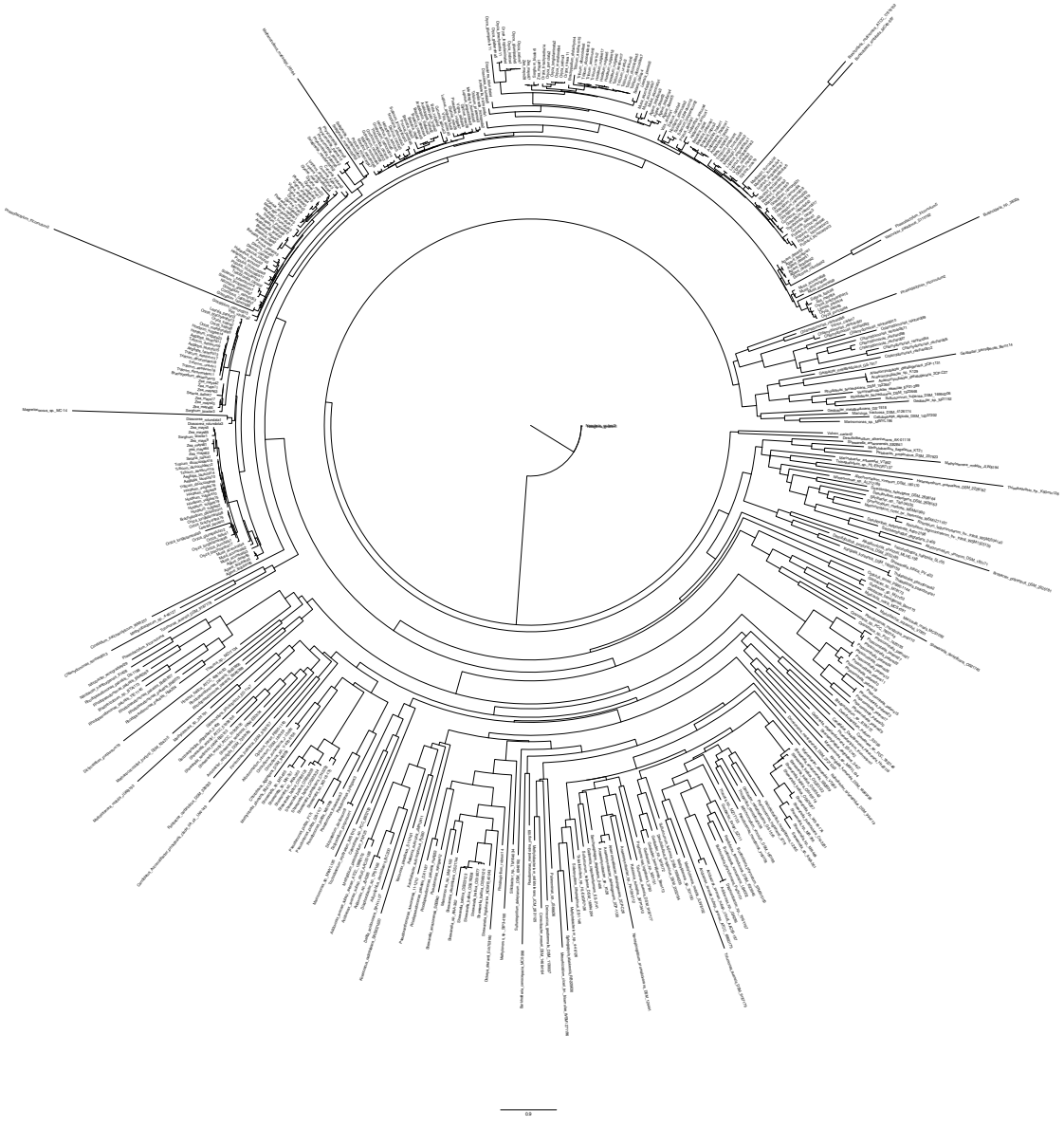


Figure 6.24.: Phylogenetic tree of the CHASE domain

Each iteration of the K-means algorithm consists of two steps: first, it associates each observation in the data set with a cluster by finding the cluster that minimizes the distance between the observation and the cluster center. In the second step, it recomputes the cluster centers by finding the mean of all data points associated with a cluster. We repeat this two-step procedure until the association of observations to clusters stops changing.

Herein, the cluster centers can be interpreted as the parameters in a generative model the algorithm assumes to be a Gaussian mixture. In each iterative step the K-means algorithm updates these parameters to maximize the probability for the model to generate the observed data. This makes K-means an instance of an expectation maximizer [28].

The advantage of the K-means algorithm is that it is efficient, i.e., its complexity is linear in the number of clusters. Also, the algorithm is guaranteed to converge and recovers clusters faithfully if each cluster is compactly distributed [103]. Its downside is that it finds only a local optimum, leaving the quality of the clustering highly dependent on the choice of the initial encoder. Also, we must either know in advance or guess the number of clusters K to find [62].

6.6.1. Methods

Here, we adapt the distributed K-means algorithm[9] presented by Dhillon and Modha [55]. This algorithm uses message passing to synchronize several processes each responsible for a disjunct partition of the original data set. To adapt this algorithm for Cuneiform we employ data dependencies where the algorithm employs message passing. This results in a functional description of the algorithm where message passing is implicit in the data dependencies between the function calls.

Other methods for speeding up the K-means algorithm have been proposed. E.g., fast and exact K-means (FEKM) [120] performs K-means on a single core but manages to never hold the whole data set in memory by sampling and clever selection of pivotal data. LSP2P and USP2P [47] depend on peer-to-peer networks to approximate K-means. They eliminate the need to synchronize the state of the whole algorithm, allowing communication only between neighboring nodes. However, we have picked Dhillon and Modha's method because it uses data-parallelism to speed up K-means and also use a communication model that easily carries over to Cuneiform's functional style.

In distributed K-means we partition the data set into several disjunct partitions. Analogous to the first step in the original algorithm the distributed algorithm associates each observation in a partition to one of the cluster centers. Next, analogous to the second step, the algorithm recomputes the cluster centers individually for each partition. Lastly, it computes the weighted mean over all cluster centers forming the cluster centers for the next iteration. Herein, the weights make sure that imbalances in the size of the partitions do not skew the result of the algorithm. We provide a Racket library that implements the individual steps in this algorithm.[10] In Appendix C.5 we give the source code for the K-means Cuneiform workflow.

[9] `https://github.com/joergen7/distributed-k-means`
[10] `https://github.com/joergen7/k-means`

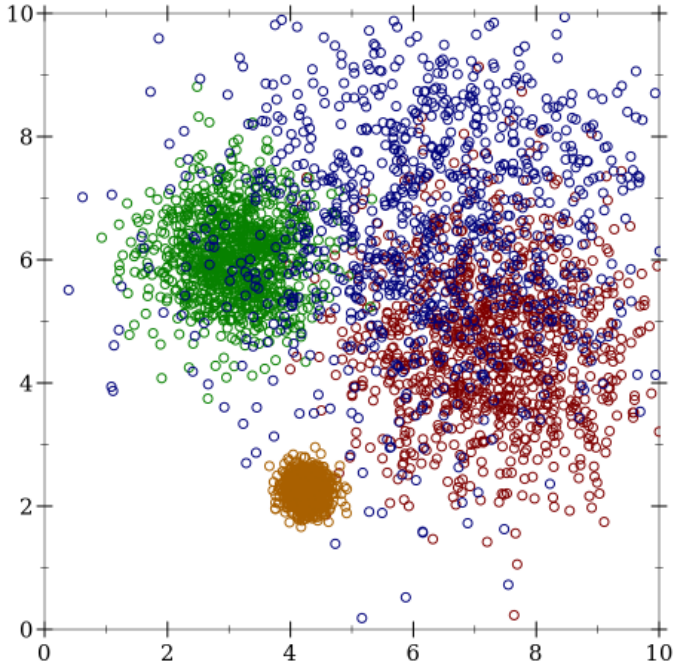


Figure 6.25.: Generated sample data with original cluster assignment

6.6.2. Results

The K-means Cuneiform workflow begins generating observations drawn from four clusters with distinct means and variances. Figure 6.25 shows the original data set and their cluster association. Dropping the cluster association, shuffling all observations, and partitioning them into ten disjunct partitions gives us the input data for the distributed K-means algorithm.

Next, we generate four random points to use as the initial cluster centers. We know that the algorithm has terminated if applying one iteration of the distributed K-means algorithm does not update cluster centers. For as long as the cluster centers change, we start a new iteration. Figure 6.26 shows how updating the cluster centers in each iteration moves the cluster centers towards their final position indicated by a cross.

Lastly, we associate each observation according to the closest cluster center. The so-derived encoder minimizes the mean square error from the cluster centers. Figure 6.27 applies this encoder function to each observation and visualizes the result. The visualizations are a byproduct of running the Cuneiform workflow.

6.6.3. Discussion

The Cuneiform implementation of the K-means algorithm demonstrates that classical results from distributed artificial intelligence can be adapted to Cuneiform. It also shows that Cuneiform, using recursion to express iteration and conditionals to express a terminal condition, is flexible enough to express distributed gradient descent and expectation

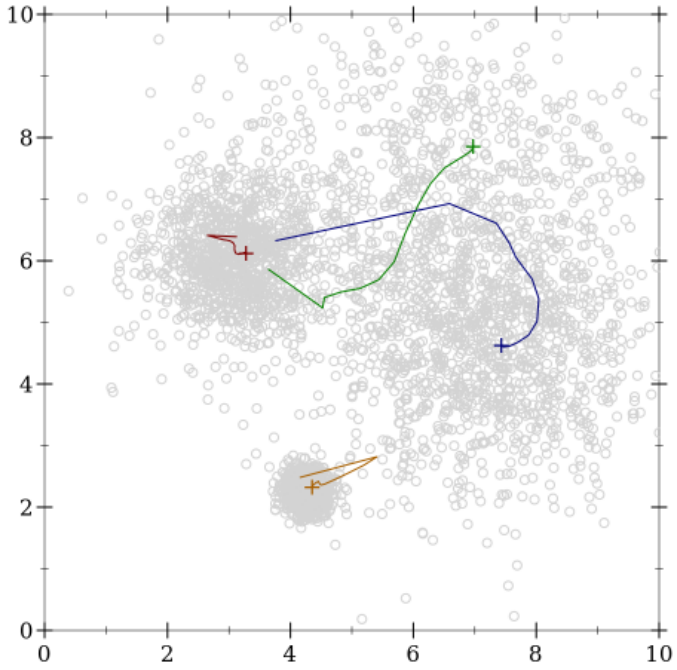


Figure 6.26.: History of k-means cluster centers from initial state to local optimum

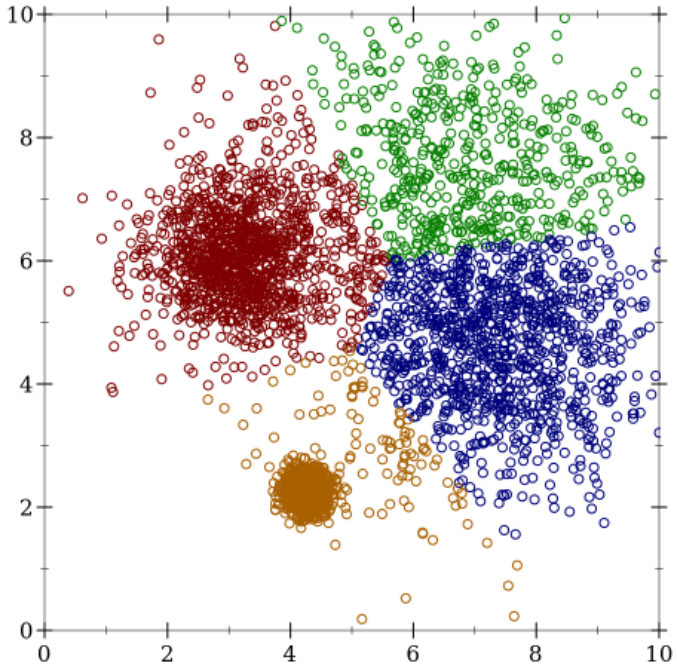


Figure 6.27.: Partitioning of the input data generated by applying k-means

maximization schemes. Cuneiform's functional interface lets the user state data dependencies directly without manually elaborating a message passing protocol.

7. Discussion

In this section we critically disuss Cuneiform. We revisit Cuneiform's language design and discuss its impact on bioinformatics use cases. We discuss the way we specify Cuneiform's syntax and semantics in this thesis and weigh it against alternative specification approaches. Next, we do the same for the way we specify Cuneiform's distributed execution environment. Lastly, we discuss the consequences of creating a distributed execution environment for Cuneiform from the specification as opposed to refitting an existing distributed execution platform to comply with the specification.

7.1. Aptitude for Bioinformatics Use Cases

Cuneiform is intended to be used for bioinformatics and next-generation sequencing use cases. However, the potential scope of a functional programming language is much larger than that. It is disputable whether the area of bioinformatics data processing actually needs the full spectrum of features that a functional programming language offers. E.g., Cuneiform offers general recursion but a weaker, more specialized feature for unbounded iteration might not only better suit the needs of bioinformatic practitioners but would also unlock more ways to reason about workflows.

Another aspect is the form of Cuneiform being a textual programming language with a dedicated syntax. Other workflow systems like Taverna or KNIME offer graphical user interfaces which are easy to pick up especially for beginners. Workflow languages like Common Workflow Language or Pegasus DAX come in the form of data serialization languages where users apply a very simple syntax. Workflow languages like SciPipe or Nextflow build on top of existing programming languages, Go and Groovy, that bioinformatic practitioners may have prior experience with.

So, while Cuneiform introduces only a small set of syntactic constructs, it does not benefit from potential user's prior familiarity with existing systems and it requires users to embrace a functional programming style.

7.2. Separation of Language Semantics and Distributed System

We have given the definition of a system for running Cuneiform workflows in two parts: first, we have given the Cuneiform language definition as an abstract syntax with a reduction semantics, and second, we have defined a suitable distributed execution environment using Petri nets. This separation allows us to separate the meaning of the workflow language and the services providing distribution, scheduling, caching, etc.

This separation is, however, uncommon in throughout the literature about scientific workflows. Most of the literature, e.g., discussing the semantics of Taverna, attempts to define the behavior of a workflow system within a single formal framework.

Despite the advantages that come with separating the concerns of meaning and communication, introducing two formal frameworks obliges the reader to become familiar with two formal frameworks instead of one. Also, this separation must introduce an interface that connects both definitions which also needs to be studied. Lastly, potential optimization schemes that need to cross the interface between both definitions are necessarily obscured.

7.3. Type System

In Chapter 3.3 we introduce Cuneiform's type system which is a statically checked type system based on simple types without capabilities for inference. Static type checking is especially useful in workflow applications because workflow execution often takes hours or days which makes finding errors upfront important. We use simple types because we can express and type practical workflows with them. In a general-purpose programming language a simple type system would be too explicit and inflexible, however in a scientific workflow application it is a pragmatic compromise that allows rigid type checking while providing just enough flexibility to express useful workflows.

The simple type system offers no inference. This means that users need to annotate each variable used in the workflow with its according type. This way the user double-checks every expression in a workflow. The type checker points out any inconsistency between the program and its type annotation. This way, the type system has a dialogue with the user via type error messages.

Thus, we designed Cuneiform's type system to be static, simple, and with no inference. However, all three design decisions can be challenged. Checking types dynamically would result in a more flexible language. Type systems that are more advanced than simple types, e.g., type systems that include subtyping, would result in a better method of record-handling. Subtyping would also enable a more object-oriented style. Lastly, type inference would lead to more concise workflows improving readability.

As it is, the user cannot extend Cuneiform's type system. There is no way to define new types like, e.g., in Swift. The possibility to define type aliases would also benefit readability. In contrast, as implemented, all types need to be spelled out in detail.

7.4. Macro System

In Section 3.6 we define two basic macros adding the `let` and `letrec` forms to Cuneiform's abstract language. We need these forms to bridge the gap between Cuneiform's surface syntax and its abstract syntax. In the implementation both forms are expanded by Cuneiform's parser, thereby omitting a macro expansion step in its own right.

Adding more derived forms, e.g., for list and record manipulation, would make Cuneiform more flexible. Also, giving the user the means to extend Cuneiform's macro system

would widen Cuneiform's potential application areas.

7.5. Petri Nets for Modeling Distributed System

In Chapter 4 we have introduced a Petri net model of a distributed execution environment capable of running Cuneiform workflows. While modeling the execution environment with Petri nets has proven sufficient to express the basic features of a distributed execution environment and has proven practical enough to derive an implementation from this model, Petri nets are not the only modeling framework for distributed systems.

Alternatively, we could have applied a process calculus like the pi-calculus, CCS, or CSP to model the distributed execution environment. This would also have been more in line with the modeling efforts existing for Taverna.

7.6. Erlang-Based Distributed Execution Environment

In Chapter 4 we have introduced a distributed execution environment for running Cuneiform workflows. This distributed execution environment uses *gen_pnet* to progress from the previously presented Petri net model to an actual implementation. gen_pnet is a library that takes a Petri net and produces an Erlang process that complies with the given Petri net. gen_pnet provides a thin[1] compatibility layer to map Petri nets to Erlang. When such a process interacts with its environment, i.e., when it starts or stops, when it communicates with other processes, or when it detects the failure of another process, we express this activity using only basic Erlang features. Thus, the distributed execution environment we present in this thesis is only very basic. It lacks advanced features like a scheduler that uses knowledge about the workflow structure or the underlying resources available to computers like main memory, input/output, or network bandwidth. Also, the distributed execution environment we present defers all opportunities to speed up data transfer to its underlying distributed file system. In addition, the distributed execution environment manages the limited resources available in a computer by assigning work to a fixed number of slots per computer. This basic approach can lead to over- or undersubscription of some critical resource and it does not dynamically react to changes in the demand of resources inherent in a workflow. Instead, the number of slots available for computation remains fixed and changes only when a new computer joins the distributed execution environment or when a computer becomes unreachable.

Existing distributed execution environments like HTCondor or Hadoop address some of the aforementioned limitations. So, an alternative way to run Cuneiform workflows would be to adapt an existing distributed execution platform to run Cuneiform workflows. Hi-WAY is such an adaptation of Hadoop for various workflow languages including Cuneiform. For future work this means that we could either extend the Erlang-based distributed execution environment to include more of these advanced features or fur-

[1] gen_pnet 0.1.7 has 352 lines of code as measured with cloc

ther pursue the approach taken in Hi-WAY by reusing an existing distributed execution platform like Hadoop.

8. Conclusion

This thesis presents Cuneiform, a distributable programming language. With Cuneiform we approach scientific workflows applying the principles of programming languages and distributed systems. As a result, with Cuneiform, researchers can express applications in bioinformatics and next-generation sequencing as functional programs. Herein, independent foreign function applications run in parallel in a distributed execution environment.

The separation of the form and meaning of Cuneiform from the distribution aspects, allows us to apply established methods from programming language theory like types, abstraction with functions, recursion, or composition.

The implementation we contribute in this thesis demonstrates that Cuneiform workflows can be executed on medium-sized clusters. We show that, given a workflow with good parallelization potential, we can saturate a cluster comprising 756 compute cores.

The distributed execution environment we present provides only basic scheduling, caching, or failure recovery methods. Furthermore, we reuse the message passing and fault detection built into Erlang, a programming language for building scalable soft real-time systems. In future work we want to integrate the extensive literature on scheduling and distribution to improve the performance of Cuneiform's distributed execution environment.

Furthermore, the Cuneiform interpreter we present provides only basic programming facilities. For the sake of simplicity we omitted several data types, polymorphism, exception handling, or continuations. Additionally, we provide only a simple type system. In future work we want to exhaust more of the available research in programming languages to make more complex workflows feasible.

We demonstrate Cuneiform's utility in several applications. We present Cuneiform workflow adaptations from bioinformatics and next-generation sequencing to clarify how we use Cuneiform's language features to express important workflow patterns. Also, we demonstrate advanced features of Cuneiform, like general recursion, in a practical context.

Cuneiform's implementation consists of three separate services: (i) a language interpreter (including a parser and a type checker), (ii) a scheduler, and (iii) a worker process. We implement all three components using the Erlang programming language. The result is a distributed programming system that users drive either by providing a workflow script or by entering expressions into an interactive shell.

We detail the distributed execution environment using Petri nets. We use open place-transition nets to describe the communication patterns between the components inside both the Cuneiform interpreter and the distributed execution environment.

We detail Cuneiform's syntax and semantics using reduction semantics, a kind of operational semantics. This way, we formally specify the form that a valid Cuneiform

script has, how the script produces its result, and how the interpreter communicates with the distributed execution environment. Additionally, we provide the specification of a simple type system.

In summary, we pick up Peter Kelly's observation that the lambda calculus is an ideal model for scientific workflows. With Cuneiform we take this one step further: a functional programming language for scientific workflows.

Bibliography

[1] Martín Abadi and Michael Isard. Timely dataflow: A model. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 131–145, Cham, 2015. Springer International Publishing.

[2] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[3] Alfred V Aho, Brian W Kernighan, and Peter J Weinberger. Awk—a pattern scanning and processing language. *Software: Practice and Experience*, 9(4):267–279, 1979.

[4] Alfred V Aho, Brian W Kernighan, and Peter J Weinberger. *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[6] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 1:1–1:13, New York, NY, USA, 2012. ACM.

[7] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 423–424, June 2004.

[8] Ilkay Altintas. Distributed workflow-driven analysis of large-scale biological data using biokepler. In *Proceedings of the 2Nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*, PDAC '11, pages 41–42, New York, NY, USA, 2011. ACM.

[9] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, pages 118–132, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[10] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludaescher, and Steve Mock. Kepler: Towards a grid-enabled system for scientific workflows. In *the Workflow in Grid Systems Workshop in GGF10-The Tenth Global Grid Forum, Berlin, Germany*, 2004.

[11] Peter Amstutz, Robin Andeer, Brad Chapman, John Chilton, Michael R Crusoe, Roman Valls Guimera, Guillermo Carrasco Hernandez, Sinisa Ivkovic, Andrey Kartashov, John Kern, et al. Common workflow language, draft 3, 2016.

[12] Vivek Anantharaman and L Aravind. The chase domain: a predicted ligand-binding module in plant cytokinin receptors and other eukaryotic and bacterial receptors. *Trends in Biochemical Sciences*, 26(10):579 – 582, 2001.

[13] Krishnarao Appasani. *Epigenomics: From chromatin biology to therapeutics*. Cambridge University Press, 2012.

[14] R. M. Badia, D. Du, E. Huedo, A. Kokossis, I. M. Llorente, R. S. Montero, M. de Palol, R. Sirvent, and C. Vázquez. Integration of grid superscalar and gridway metascheduler with the drmaa ogf standard. In Emilio Luque, Tomàs Margalef, and Domingo Benítez, editors, *Euro-Par 2008 – Parallel Processing*, pages 445–455, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] Tanya Barrett and Ron Edgar. [19] gene expression omnibus: Microarray data storage, submission, retrieval, and analysis. In *DNA Microarrays, Part B: Databases and Statistics*, volume 411 of *Methods in Enzymology*, pages 352 – 369. Academic Press, 2006.

[16] Alex Bateman, Robert D. Finn, Sean R. Eddy, Jaina Mistry, and Marco Punta. Challenges in homology search: Hmmer3 and convergent evolution of coiled-coil regions. *Nucleic Acids Research*, 41(12):e121–e121, 04 2013.

[17] Alex Bateman, Andreas Heger, Erik L. L. Sonnhammer, Jaina Mistry, Jody Clements, John Tate, Kirstie Hetherington, Liisa Holm, Marco Punta, Penelope Coggill, Ruth Y. Eberhardt, Sean R. Eddy, and Robert D. Finn. Pfam: the protein families database. *Nucleic Acids Research*, 42(D1):D222–D230, 11 2013.

[18] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

[19] Jeremy M Berg, John L Tymoczko, and Lubert Stryer. Biochemistry, ; w. h, 2002.

[20] G Bruce Berriman, Ewa Deelman, John C Good, Joseph C Jacob, Daniel S Katz, Carl Kesselman, Anastasia C Laity, Thomas A Prince, Gurmeet Singh, and Mei-Hu Su. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *Proceedings of SPIE*, volume 5493, pages 221–232, 2004.

[21] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. Knime -

the konstanz information miner: Version 2.0 and beyond. *SIGKDD Explor. Newsl.*, 11(1):26–31, November 2009.

[22] Alysson Bessani, Jörgen Brandt, Marc Bux, Vinicius Cogo, Lora Dimitrova, Jim Dowling, Ali Gholami, Kamal Hakimzadeh, Micheal Hummel, Mahmoud Ismail, Erwin Laure, Ulf Leser, Jan-Eric Litton, Roxanna Martinez, Salman Niazi, Jane Reichel, and Karin Zimmermann. Biobankcloud: A platform for the secure storage, sharing, and processing of large biomedical data sets. In Fusheng Wang, Gang Luo, Chunhua Weng, Arijit Khan, Prasenjit Mitra, and Cong Yu, editors, *Biomedical Data Management and Graph Online Querying*, pages 89–105, Cham, 2016. Springer International Publishing.

[23] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *2008 Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, Nov 2008.

[24] Monika Bharti, Anju Bala, et al. Workflow management in cloud computing. *International Journal of Applied Information Systems*, 4(9), 2012.

[25] Adrian Bird. Dna methylation patterns and epigenetic memory. *Genes & development*, 16(1):6–21, 2002.

[26] Adrian Bird. Perceptions of epigenetics. *Nature*, 447(7143):396, 2007.

[27] Adrian P Bird. Cpg-rich islands and the function of dna methylation. *Nature*, 321(6067):209–213, 1986.

[28] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[29] Shawn Bowers and Bertram Ludäscher. Actor-oriented design of scientific workflows. *ER*, 2005:369–384, 2005.

[30] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2012.

[31] Jörgen Brandt, Marc Bux, and Ulf Leser. Cuneiform: a functional language for large scale scientific data analysis. In *EDBT/ICDT Workshops*, pages 7–16, 2015.

[32] Jörgen Brandt and Wolfgang Reisig. Modeling erlang processes as petri nets. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, Erlang 2018, pages 61–66, New York, NY, USA, 2018. ACM.

[33] JÖRGEN BRANDT, WOLFGANG REISIG, and ULF LESER. Computation semantics of the functional scientific workflow language cuneiform. *Journal of Functional Programming*, 27:e22, 2017.

[34] Silvia Breitinger, Ulrike Klusik, and Rita Loogen. From (sequential) haskell to (parallel) eden: An implementation point of view. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, pages 318–334, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[35] Roger Brobst, Waiman Chan, Fritz Ferstl, Andreas Haas, Daniel Templeton, and John Tollefsrud. Distributed resource management application api specification 1.0. In *Grid Forum Document GFD*, volume 22, 2004.

[36] Mihai Budiu and Seth C Goldstein. Pegasus: An efficient intermediate representation. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[37] Marc Bux, Jörgen Brandt, Carsten Lipka, Kamal Hakimzadeh, Jim Dowling, and Ulf Leser. Saasfee: scalable scientific workflow execution engine. *Proceedings of the VLDB Endowment*, 8(12):1892–1895, 2015.

[38] Marc Bux, Jörgen Brandt, Carl Witt, Jim Dowling, and Ulf Leser. Hi-way: Execution of scientific workflows on hadoop yarn. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT).*, Venice, Italy, 2017.

[39] Marc Bux, Jörgen Brandt, Carl Witt, Jim Dowling, and Ulf Leser. Hi-way: Execution of scientific workflows on hadoop yarn. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT).*, Venice, Italy, 2017.

[40] S. Callaghan, P. Maechling, E. Deelman, K. Vahi, G. Mehta, G. Juve, K. Milner, R. Graves, E. Field, D. Okaya, D. Gunter, K. Beattie, and T. Jordan. Reducing time-to-solution using distributed high-throughput mega-workflows - experiences from scec cybershake. In *2008 IEEE Fourth International Conference on eScience*, pages 151–158, Dec 2008.

[41] Junwei Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: workflow management for grid computing. In *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pages 198–205, May 2003.

[42] Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems.* " O'Reilly Media, Inc.", 2016.

[43] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1985.

[44] Sarah Cohen-Boulakia, Jiuqiang Chen, Paolo Missier, Carole Goble, Alan R. Williams, and Christine Froidevaux. Distilling structure in taverna scientific workflows: a refactoring approach. *BMC Bioinformatics*, 15(1):S12, Jan 2014.

[45] 1000 Genomes Project Consortium et al. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56, 2012.

[46] Vasa Curcin and Moustafa Ghanem. Scientific workflow systems-can one size fit all? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–9. IEEE, 2008.

[47] S. Datta, C. Giannella, and H. Kargupta. Approximate distributed k-means clustering over a peer-to-peer network. *IEEE Transactions on Knowledge and Data Engineering*, 21(10):1372–1388, Oct 2009.

[48] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: Challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1345–1350, New York, NY, USA, 2008. ACM.

[49] David De Roure, Carole A Goble, Graham Klyne, Marco Roos, Kristina M Hettne, José Enrique Ruiz, Raúl Palma, José Manuél Gómez-Pérez, Paolo Missier, and Khalid Belhajjame. Towards the preservation of scientific workflows. In *iPRES*, 2011.

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[51] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T. H. Jordan, C. Kesselman, P. Maechling, J. Mehringer, G. Mehta, D. Okaya, K. Vahi, and L. Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 14–14, Dec 2006.

[52] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In Marios D. Dikaiakos, editor, *Grid Computing*, pages 11–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[53] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: The montage example. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[54] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17 – 35, 2015.

[55] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In Mohammed J. Zaki and Ching-Tien Ho, editors, *Large-Scale Parallel Data Mining*, pages 245–260, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[56] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[57] Harry C Dietz. New therapeutic approaches to mendelian disorders. *New England Journal of Medicine*, 363(9):852–863, 2010.

[58] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proc. VLDB Endow.*, 5(12):2014–2015, August 2012.

[59] Bruno Domon and Ruedi Aebersold. Challenges and opportunities in proteomics data analysis. *Molecular & Cellular Proteomics*, 5(10):1921–1926, 2006.

[60] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[61] Alexei Drummond and Allen G. Rodrigo. Reconstructing Genealogies of Serial Samples Under the Assumption of a Molecular Clock Using Serial-Sample UPGMA. *Molecular Biology and Evolution*, 17(12):1807–1815, 12 2000.

[62] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[63] S R Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 10 1998.

[64] Sean R Eddy. Hidden markov models. *Current Opinion in Structural Biology*, 6(3):361 – 365, 1996.

[65] Robert C. Edgar. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5(1):113, Aug 2004.

[66] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 03 2004.

[67] Ron Edgar, Michael Domrachev, and Alex E. Lash. Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic Acids Research*, 30(1):207–210, 01 2002.

[68] David Elliott and Michael Ladomery. *Molecular biology of RNA*. Oxford University Press, 2017.

[69] Joshua Elliott, Ian Foster, Kenneth Judd, Elisabeth Moyer, and Todd Munson. Cim-earth: Framework and case study. *The BE Journal of Economic Analysis & Policy*, 10(2), 2010.

[70] Joshua Elliott, Ian Foster, Kenneth Judd, Elisabeth Moyer, Todd Munson, et al. Cim-earth: Community integrated model of economic and resource trajectories for humankind. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2010.

[71] Joshua Elliott, Ian Foster, Samuel Kortum, Todd Munson, Fernando Pérez Cervantes, and David Weisbach. Trade and carbon taxes. *American Economic Review*, 100(2):465–69, May 2010.

[72] W Gregory Feero, Alan E Guttmacher, and Francis S Collins. Genomic medicine—an updated primer. *New England Journal of Medicine*, 362(21):2001–2011, 2010.

[73] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[74] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[75] Kathleen M. Fisch, Tobias Meißner, Louis Gioia, Jean-Christophe Ducom, Tristan M. Carland, Salvatore Loguercio, and Andrew I. Su. Omics pipe: a community-based framework for reproducible multi-omics data analysis. *Bioinformatics*, 31(11):1724–1728, 2015.

[76] Glenn Fowler. A case for make. *Software: Practice and Experience*, 20(S1):S35–S46, 1990.

[77] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[78] Juliana Freire, Cláudio T. Silva, Steven P. Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. Managing rapidly-evolving scientific workflows. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, pages 10–18, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[79] Olivier Gascuel and Si Quang Le. An Improved General Amino Acid Replacement Matrix. *Molecular Biology and Evolution*, 25(7):1307–1320, 03 2008.

[80] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: The pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[81] W. Gentzsch. Sun grid engine: towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[82] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor,

et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10):1451–1455, 2005.

[83] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, Dec 2007.

[84] T. Glatard and J. Montagnat. Implementation of turing machines with the scufl data-flow language. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 663–668, May 2008.

[85] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole Goble. Composing different models of computation in kepler and ptolemy ii. In Yong Shi, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2007*, pages 182–190, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[86] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, Aug 2010.

[87] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):1, 2010.

[88] Loyal A Goff, Cole Trapnell, and David Kelley. Cummerbund: visualization and exploration of cufflinks high-throughput sequencing data. *R package version*, 2(0), 2012.

[89] Manfred G. Grabherr, Brian J. Haas, Moran Yassour, Joshua Z. Levin, Dawn A. Thompson, Ido Amit, Xian Adiconis, Lin Fan, Raktima Raychowdhury, Qiandong Zeng, Zehua Chen, Evan Mauceli, Nir Hacohen, Andreas Gnirke, Nicholas Rhind, Federica di Palma, Bruce W. Birren, Chad Nusbaum, Kerstin Lindblad-Toh, Nir Friedman, and Aviv Regev. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nature Biotechnology*, 29:644, May 2011.

[90] Gregory E Graham, Dave Evans, and Iain Bertram. Mcrunjob: A high energy physics workflow planner for grid production processing. *arXiv preprint cs/0305063*, 2003.

[91] Robert Graves, Thomas H. Jordan, Scott Callaghan, Ewa Deelman, Edward Field, Gideon Juve, Carl Kesselman, Philip Maechling, Gaurang Mehta, Kevin Milner, David Okaya, Patrick Small, and Karan Vahi. Cybershake: A physics-based seismic hazard model for southern california. *Pure and Applied Geophysics*, 168(3):367–381, Mar 2011.

[92] Anthony JF Griffiths. *An introduction to genetic analysis*. Macmillan, 2005.

[93] Ilan Gronau and Shlomo Moran. Optimal implementations of upgma and other common clustering algorithms. *Information Processing Letters*, 104(6):205 – 210, 2007.

[94] Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. Grid-flow: a grid-enabled scientific workflow system with a petri-net-based interface. *Concurrency and Computation: Practice and Experience*, 18(10):1115–1140, 2006.

[95] Roman Valls Guimera. bcbio-nextgen: Automated, distributed next-gen sequencing pipeline. *EMBnet. journal*, 17(B):p–30, 2012.

[96] Stéphane Guindon, Frédéric Delsuc, Jean-François Dufayard, and Olivier Gascuel. *Estimating Maximum Likelihood Phylogenies with PhyML*, pages 113–137. Humana Press, Totowa, NJ, 2009.

[97] Stéphane Guindon, Jean-François Dufayard, Vincent Lefort, Maria Anisimova, Wim Hordijk, and Olivier Gascuel. New Algorithms and Methods to Estimate Maximum-Likelihood Phylogenies: Assessing the Performance of PhyML 3.0. *Systematic Biology*, 59(3):307–321, 05 2010.

[98] Stéphane Guindon and Olivier Gascuel. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology*, 52(5):696–704, 10 2003.

[99] Rama. Gullapalli, Ketaki. Desai, Lucas. Santana-Santos, Jeffrey. Kant, and Michael. Becich. Next generation sequencing in clinical medicine: Challenges and lessons for pathology and biomedical informatics. *Journal of Pathology Informatics*, 3(1):40, 2012.

[100] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[101] Kasper D. Hansen, Benjamin Langmead, and Rafael A. Irizarry. Bsmooth: from whole genome bisulfite sequencing reads to differentially methylated regions. *Genome Biology*, 13(10):R83, Oct 2012.

[102] Helen E Harrison, Stephen P Schaefer, and Terry S Yoo. Rtools: Tools for software management in a distributed computing environment. In *Proceedings of the Summer USENIX Conference*, pages 85–93, 1988.

[103] Simon Haykin. *Neural networks*, volume 2. Prentice hall New York, 1994.

[104] Monika Heiner, David Gilbert, and Robin Donaldson. Petri nets for systems and synthetic biology. In Marco Bernardo, Pierpaolo Degano, and Gianluigi Zavattaro, editors, *Formal Methods for Computational Systems Biology*, pages 215–264, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[105] J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente. Developing grid-aware applications with drmaa on globus-based grids. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, pages 429–435, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[106] Carl Herrmann, Denis Thieffry, Matthieu Defrance, Morgane Thomas-Chollier, Olivier Sand, and Jacques van Helden. RSAT peak-motifs: motif analysis in full-size ChIP-seq datasets. *Nucleic Acids Research*, 40(4):e31–e31, 12 2011.

[107] Anthony JG Hey and Anne E Trefethen. The data deluge: An e-science perspective, 2003.

[108] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[109] Alexander Heyl, Klaas Wulfetange, Birgit Pils, Nicola Nielsen, Georgy A. Romanov, and Thomas Schmülling. Evolutionary proteomics identifies amino acids essential for ligand-binding of the cytokinin receptor chase domain. *BMC Evolutionary Biology*, 7(1):62, Apr 2007.

[110] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. Dfl: A dataflow language based on petri nets and nested relational calculus. *Information Systems*, 33(3):261 – 284, 2008.

[111] Jan Hidders and Jacek Sroka. Towards a calculus for collection-oriented scientific workflows with side effects. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems: OTM 2008*, pages 374–391, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[112] Ian H Holmes and Christopher J Mungall. Biomake: a gnu make-compatible utility for declarative workflow management. *Bioinformatics*, 33(21):3502–3504, 2017.

[113] Jin Hu, Jun Zhu, and HM Xu. Methods of constructing core collections by stepwise clustering with three sampling strategies based on the genotypic values of crops. *Theoretical and Applied Genetics*, 101(1-2):264–268, 2000.

[114] Kathy L Hudson. Genomics, health care, and society. *New England Journal of Medicine*, 365(11):1033–1041, 2011.

[115] John P Huelsenbeck and Fredrik Ronquist. Mrbayes: Bayesian inference of phylogenetic trees. *Bioinformatics*, 17(8):754–755, 2001.

[116] M. N. Huhns and M. P. Singh. Service-oriented computing: key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, Jan 2005.

[117] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl 2):W729–W732, 2006.

[118] Bernd Jagla, Bernd Wiswedel, and Jean-Yves Coppée. Extending knime for next-generation sequencing data analysis. *Bioinformatics*, 27(20):2907–2909, 2011.

[119] Jia Yu, R. Buyya, and Chen Khong Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *First International Conference on e-Science and Grid Computing (e-Science'05)*, pages 8 pp.–147, July 2005.

[120] Ruoming Jin, Anjan Goswami, and Gagan Agrawal. Fast and exact out-of-core and distributed k-means clustering. *Knowledge and Information Systems*, 10(1):17–40, Jul 2006.

[121] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[122] Peter A. Jones and Daiya Takai. The role of dna methylation in mammalian epigenetics. *Science*, 293(5532):1068–1070, 2001.

[123] Laurent Jourdren, Maria Bernard, Marie-Agnès Dillies, and Stéphane Le Crom. Eoulsan: a cloud computing-based framework facilitating high throughput sequencing analyses. *Bioinformatics*, 28(11):1542–1543, 2012.

[124] G. Juve, M. Rynge, E. Deelman, J. S. Vöckler, and G. B. Berriman. Comparing futuregrid, amazon ec2, and open science grid for scientific workflows. *Computing in Science Engineering*, 15(4):20–29, July 2013.

[125] Gideon Juve and Ewa Deelman. *Scientific Workflows in the Cloud*, pages 71–91. Springer London, London, 2011.

[126] G. Kahn. Natural semantics. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87*, pages 22–39, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[127] Peter M Kelly. *Applying functional programming theory to the design of workflow engines*. PhD thesis, The University of Adelaide, 2011.

[128] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Lambda calculus as a workflow model. *Concurrency and Computation: Practice and Experience*, 21(16):1999–2017, 2009.

[129] Peter M. Kelly, Paul D. Coddington, and Andrew L. Wendelborn. Lambda calculus as a workflow model. *Concurrency and Computation: Practice and Experience*, 21(16):1999–2017, 2009.

[130] Daehwan Kim, Ben Langmead, and Steven L Salzberg. Hisat: a fast spliced aligner with low memory requirements. *Nature methods*, 12(4):357, 2015.

[131] Teri E Klein, Jeffrey T Chang, Mildred K Cho, Katrina L Easton, Ray Fergerson, Micheal Hewett, Zhen Lin, Y Liu, S Liu, DE Oliver, et al. Integrating genotype

and phenotype information: an overview of the pharmgkb project. *The pharmacogenomics journal*, 1(3):167, 2001.

[132] Daniel C. Koboldt, Ken Chen, Todd Wylie, David E. Larson, Michael D. McLellan, Elaine R. Mardis, George M. Weinstock, Richard K. Wilson, and Li Ding. Varscan: variant detection in massively parallel sequencing of individual and pooled samples. *Bioinformatics*, 25(17):2283–2285, 2009.

[133] Daniel C. Koboldt, David E. Larson, and Richard K. Wilson. Using varscan 2 for germline variant calling and somatic mutation detection. *Current Protocols in Bioinformatics*, 44(1):15.4.1–15.4.17, 2013.

[134] Daniel C Koboldt, Qunyuan Zhang, David E Larson, Dong Shen, Michael D McLellan, Ling Lin, Christopher A Miller, Elaine R Mardis, Li Ding, and Richard K Wilson. Varscan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome research*, 2012.

[135] Johannes Köster and Sven Rahmann. Building and Documenting Workflows with Python-Based Snakemake. In Sebastian Böcker, Franziska Hufsky, Kerstin Scheubert, Jana Schleicher, and Stefan Schuster, editors, *German Conference on Bioinformatics 2012*, volume 26 of *OpenAccess Series in Informatics (OASIcs)*, pages 49–56, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[136] Johannes Köster and Sven Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[137] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357, 2012.

[138] Ben Langmead, Michael C Schatz, Jimmy Lin, Mihai Pop, and Steven L Salzberg. Searching for snps with cloud computing. *Genome biology*, 10(11):R134, 2009.

[139] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, Mar 2009.

[140] Jeremy Leipzig. A review of bioinformatic pipeline frameworks. *Briefings in Bioinformatics*, 18(3):530–536, 2017.

[141] Arthur Lesk. *Introduction to genomics*. Oxford University Press, 2017.

[142] John R Levine, John Mason, John R Levine, John R Levine, Paul Levine, Tony Mason, and Doug Brown. *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

[143] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[144] C. Lin and S. Lu. Scheduling scientific workflows elastically for cloud computing. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 746–747, July 2011.

[145] Pierre Lindenbaum, Solena Le Scouarnec, Vincent Portero, and Richard Redon. Knime4bio: a set of custom nodes for the interpretation of next-generation sequencing data with knime†. *Bioinformatics*, 27(22):3200–3201, 2011.

[146] Michel J Litzkow, Miron Livny, and Matt W Mutka. Condor-a hunter of idle workstations. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1987.

[147] Xiangtao Liu, Shizhong Han, Zuoheng Wang, Joel Gelernter, and Bao-Zhu Yang. Variant callers for next-generation sequencing data: a comparison study. *PloS one*, 8(9):e75619, 2013.

[148] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[149] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel functional programming in eden. *Journal of Functional Programming*, 15(03):431–475, 2005.

[150] Yi-Fan Lu, David B. Goldstein, Misha Angrist, and Gianpiero Cavalleri. Personalized medicine and human genetic diversity. *Cold Spring Harbor Perspectives in Medicine*, 4(9), 2014.

[151] Bertram Ludäscher and Ilkay Altintas. On providing declarative design and programming constructs for scientific workflows based on process networks, 2003.

[152] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[153] Tatsuya Maeda, Susannah M Wurgler-Murphy, and Haruo Saito. A two-component system that regulates an osmosensing map kinase cascade in yeast. *Nature*, 369(6477):242, 1994.

[154] Teri A Manolio. Genomewide association studies and assessment of the risk of disease. *New England Journal of Medicine*, 363(2):166–176, 2010.

[155] Elaine R Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, 24(3):133–141, 2008.

[156] Vivien Marx. The big challenges of big data. *Nature*, 498:255, June 2013.

[157] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. Adam: Genomics formats

and processing patterns for cloud scale computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-207*, 2013.

[158] Mark I McCarthy. Genomics, type 2 diabetes, and obesity. *New England Journal of Medicine*, 363(24):2339–2350, 2010.

[159] Michael McLennan, Steven Clark, Ewa Deelman, Mats Rynge, Karan Vahi, Frank McKenna, Derrick Kearney, and Carol Song. Bringing scientific workflow to the masses via pegasus and hubzero. *parameters*, 13:14, 2013.

[160] Timothy McPhillips, Shawn Bowers, and Bertram Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In Ulf Leser, Felix Naumann, and Barbara Eckman, editors, *Data Integration in the Life Sciences*, pages 248–263, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[161] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[162] Saverio Minucci and Pier Giuseppe Pelicci. Histone deacetylase inhibitors and the promise of epigenetic (and more) treatments for cancer. *Nature Reviews Cancer*, 6(1):38, 2006.

[163] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991.

[164] Ali Mortazavi, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature methods*, 5(7):621, 2008.

[165] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[166] Hamid Mushtaq, Frank Liu, Carlos Costa, Gang Liu, Peter Hofstee, and Zaid Al-Ars. Sparkga: A spark framework for cost effective, fast and accurate dna analysis at scale. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ACM-BCB '17, pages 148–157, New York, NY, USA, 2017. ACM.

[167] Kevin S. Myers, Huihuang Yan, Irene M. Ong, Dongjun Chung, Kun Liang, Frances Tran, Sündüz Keleş, Robert Landick, and Patricia J. Kiley. Genome-scale analysis of escherichia coli fnr reveals complex features of transcription factor binding. *PLOS Genetics*, 9(6):1–24, 06 2013.

[168] Mike A. Nalls, Nathan Pankratz, Christina M. Lill, Chuong B. Do, Dena G. Hernandez, Mohamad Saad, Anita L. DeStefano, Eleanna Kara, Jose Bras, Manu

Sharma, Claudia Schulte, Margaux F. Keller, Sampath Arepalli, Christopher Letson, Connor Edsall, Hreinn Stefansson, Xinmin Liu, Hannah Pliner, Joseph H. Lee, Rong Cheng, International Parkinson's Disease Genomics Consortium (IPDGC), Parkinson's Study Group (PSG) Parkinson's Research: The Organized GENetics Initiative (PROGENI), 23andMe, GenePD, NeuroGenetics Research Consortium (NGRC), Hussman Institute of Human Genomics (HIHG), The Ashkenazi Jewish Dataset Investigator, Cohorts for Health, Aging Research in Genetic Epidemiology (CHARGE), North American Brain Expression Consortium (NABEC), United Kingdom Brain Expression Consortium (UKBEC), Greek Parkinson's Disease Consortium, Alzheimer Genetic Analysis Group, M. Arfan Ikram, John P. A. Ioannidis, Georgios M. Hadjigeorgiou, Joshua C. Bis, Maria Martinez, Joel S. Perlmutter, Alison Goate, Karen Marder, Brian Fiske, Margaret Sutherland, Georgia Xiromerisiou, Richard H. Myers, Lorraine N. Clark, Kari Stefansson, John A. Hardy, Peter Heutink, Honglei Chen, Nicholas W. Wood, Henry Houlden, Haydeh Payami, Alexis Brice, William K. Scott, Thomas Gasser, Lars Bertram, Nicholas Eriksson, Tatiana Foroud, and Andrew B. Singleton. Large-scale meta-analysis of genome-wide association data identifies six new risk loci for parkinson's disease. *Nature Genetics*, 46:989, July 2014.

[169] M Nei and N Saitou. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 07 1987.

[170] Tung Nguyen, Weisong Shi, and Douglas Ruden. Cloudaligner: A fast and full-featured mapreduce based tool for sequence mapping. *BMC research notes*, 4(1):171, 2011.

[171] Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. Biopig: a hadoop-based analytic toolkit for large-scale sequence data. *Bioinformatics*, 29(23):3014–3019, 2013.

[172] Christopher J O'donnell and Elizabeth G Nabel. Genomics of cardiovascular disease. *New England Journal of Medicine*, 365(22):2098–2109, 2011.

[173] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[174] Andrea Ordanini and Paolo Pasini. Service co-production and value co-creation: The case for a service-oriented architecture (soa). *European Management Journal*, 26(5):289 – 297, 2008.

[175] Qun Pan, Ofer Shai, Leo J Lee, Brendan J Frey, and Benjamin J Blencowe. Deep surveying of alternative splicing complexity in the human transcriptome by high-throughput sequencing. *Nature genetics*, 40(12):1413, 2008.

[176] Ram Vinay Pandey and Christian Schlötterer. Distmap: a toolkit for distributed short read mapping on a hadoop cluster. *PLoS One*, 8(8):e72614, 2013.

[177] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, Nov 2007.

[178] Peter J Park. Chip–seq: advantages and challenges of a maturing technology. *Nature Reviews Genetics*, 10(10):669, 2009.

[179] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[180] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[181] Elizabeth Pennisi. Will computers crash genomics? *Science*, 331(6018):666–668, 2011.

[182] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[183] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[184] Birgit Pils and Alexander Heyl. Unraveling the evolution of cytokinin signaling. *Plant Physiology*, 151(2):782–791, 2009.

[185] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. Seal: a distributed short read mapping and duplicate removal tool. *Bioinformatics*, 27(15):2159–2160, 2011.

[186] Gordon Plotkin. An operational semantics for csp. In A. Salwicki, editor, *Logics of Programs and Their Applications*, pages 250–252, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[187] Gordon D Plotkin. A structural approach to operational semantics, 1981.

[188] R.F. Pointon, P.W. Trinder, and H.-W. Loidl. The design and implementation of glasgow distributed haskell. In Markus Mohnen and Pieter Koopman, editors, *Implementation of Functional Languages*, pages 53–70, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[189] Aaron R. Quinlan. Bedtools: The swiss-army tool for genome feature analysis. *Current Protocols in Bioinformatics*, 47(1):11.12.1–11.12.34, 2014.

[190] Aaron R. Quinlan and Ira M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 01 2010.

[191] Fidel Ramírez, Sarah Diehl, Thomas Manke, Friederike Dündar, and Björn A. Grüning. deepTools: a flexible platform for exploring deep-sequencing data. *Nucleic Acids Research*, 42(W1):W187–W191, 05 2014.

[192] Aharon Razin. *DNA Methylation Patterns: Formation and Biological Functions*, pages 127–146. Springer New York, New York, NY, 1984.

[193] Wolfgang Reisig. *Systementwurf mit Netzen*. Springer-Verlag, 2013.

[194] Wolfgang Reisig. *Understanding petri nets: modeling techniques, analysis methods, case studies*. Springer, 2013.

[195] Jason A. Reuter, Damek V. Spacek, and Michael P. Snyder. High-throughput sequencing technologies. *Molecular Cell*, 58(4):586 – 597, 2015.

[196] Boris Reva, Yevgeniy Antipin, and Chris Sander. Predicting the functional impact of protein mutations: application to cancer genomics. *Nucleic Acids Research*, 39(17):e118, 2011.

[197] James T Robinson, Helga Thorvaldsdóttir, Wendy Winckler, Mitchell Guttman, Eric S Lander, Gad Getz, and Jill P Mesirov. Integrative genomics viewer. *Nature biotechnology*, 29(1):24, 2011.

[198] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[199] Rizos Sakellariou, Henan Zhao, and Ewa Deelman. Mapping workflows on grid resources: experiments with the montage workflow. *Grids, P2P and Services Computing*, pages 119–132, 2010.

[200] Eric E Schadt, Michael D Linderman, Jon Sorenson, Lawrence Lee, and Garry P Nolan. Computational solutions to large-scale data management and analysis. *Nature reviews genetics*, 11(9):647, 2010.

[201] Michael C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[202] Christopher Schiefer, Marc Bux, Joergen Brandt, Clemens Messerschmidt, Knut Reinert, Dieter Beule, and Ulf Leser. Portability of scientific workflows in ngs data analysis: A case study, 2020.

[203] Matthew B. Scholz, Chien-Chi Lo, and Patrick SG Chain. Next generation sequencing and bioinformatic bottlenecks: the current state of metagenomic data analysis. *Current Opinion in Biotechnology*, 23(1):9 – 15, 2012. Analytical biotechnology.

[204] André Schumacher, Luca Pireddu, Matti Niemenmaa, Aleksi Kallio, Eija Korpelainen, Gianluigi Zanetti, and Keijo Heljanko. Seqpig: simple and scalable scripting for large sequencing data sets in hadoop. *Bioinformatics*, 30(1):119–120, 2014.

[205] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature Biotechnology*, 26:1135, October 2008.

[206] Y. Simmhan, R. Barga, C. v. Ingen, E. Lazowska, and A. Szalay. Building the trident scientific workflow workbench for data management in the cloud. In *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 41–50, Oct 2009.

[207] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, September 2005.

[208] Robert R. Sokal and F. James Rohlf. The comparison of dendrograms by objective methods. *Taxon*, 11(2):33–40, 1962.

[209] Jacek Sroka and Jan Hidders. Towards a formal semantics for the process model of the taverna workbench. part i. *Fundamenta Informaticae*, 92(3):279–299, 2009.

[210] Jacek Sroka and Jan Hidders. Towards a formal semantics for the process model of the taverna workbench. part ii. *Fundamenta Informaticae*, 92(4):373–396, 2009.

[211] Jacek Sroka, Jan Hidders, Paolo Missier, and Carole Goble. A formal semantics for the taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6):490 – 508, 2010. Special Issue: Scientific Workflow 2009.

[212] Ian J Taylor, Ewa Deelman, Dennis B Gannon, Matthew Shields, et al. *Workflows for e-Science: scientific workflows for grids*, volume 1. Springer, 2007.

[213] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, August 1976.

[214] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. *Grid computing: Making the global infrastructure a reality*, pages 299–335, 2003.

[215] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[216] Morgane Thomas-Chollier, Elodie Darbo, Carl Herrmann, Matthieu Defrance, Denis Thieffry, and Jacques Van Helden. A complete workflow for the analysis of full-size chip-seq (and similar) data sets using peak-motifs. *Nature protocols*, 7(8):1551, 2012.

[217] Julie D. Thompson, Toby. J. Gibson, and Des G. Higgins. Multiple sequence alignment using clustalw and clustalx. *Current Protocols in Bioinformatics*, 00(1):2.3.1–2.3.22, 2003.

[218] Helga Thorvaldsdóttir, James T. Robinson, and Jill P. Mesirov. Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in Bioinformatics*, 14(2):178–192, 04 2012.

[219] Ali Torkamani, Phillip Pham, Ondrej Libiger, Vikas Bansal, Guangfa Zhang, Ashley Scott-Van Zeeland, Ryan Tewhey, Eric Topol, and Nicholas Schork. Clinical implications of human population differences in genome-wide rates of functional genotypes. *Frontiers in Genetics*, 3:211, 2012.

[220] Seved Torstendahl. Open telecom platform. *Ericsson Review(English Edition)*, 74(1):14–23, 1997.

[221] Cole Trapnell, Lior Pachter, and Steven L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–1111, 03 2009.

[222] Cole Trapnell, Adam Roberts, Loyal Goff, Geo Pertea, Daehwan Kim, David R Kelley, Harold Pimentel, Steven L Salzberg, John L Rinn, and Lior Pachter. Differential gene and transcript expression analysis of rna-seq experiments with tophat and cufflinks. *Nature protocols*, 7(3):562, 2012.

[223] P. Troger, H. Rajic, A. Haas, and P. Domagalski. Standardization of an api for distributed resource management systems. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 619–626, May 2007.

[224] Peter Tröger, Roger Brobst, Daniel Gruber, Mariusz Mamonski, and Daniel Templeton. Distributed resource management application api version 2 (drmaa), 2012.

[225] D. Turi, P. Missier, C. Goble, D. D. Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 441–448, Dec 2007.

[226] Daniele Turi, Paolo Missier, Carole Goble, David De Roure, and Tom Oinn. Taverna workflows: Syntax and semantics. In *e-Science and Grid Computing, IEEE International Conference on*, pages 441–448. IEEE, 2007.

[227] Bryan M. Turner. Histone acetylation and an epigenetic code. *BioEssays*, 22(9):836–845, 2000.

[228] W. M. P. van der Aalst. Verification of workflow nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, pages 407–426, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[229] W. M. P. VAN DER AALST. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

[230] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, May 2011.

[231] W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245 – 275, 2005.

[232] Andrey Vasin, Sergey Klotchenko, and Ludmila Puchkova. Phylogenetic analysis of six-domain multi-copper blue proteins. *PLoS currents*, 5, 2013.

[233] Axel Visel, Matthew J Blow, Zirong Li, Tao Zhang, Jennifer A Akiyama, Amy Holt, Ingrid Plajzer-Frick, Malak Shoukry, Crystal Wright, Feng Chen, et al. Chip-seq accurately predicts tissue-specific activity of enhancers. *Nature*, 457(7231):854, 2009.

[234] David G Wang, Jian-Bing Fan, Chia-Jen Siao, Anthony Berno, Peter Young, Ron Sapolsky, Ghassan Ghandour, Nancy Perkins, Ellen Winchester, Jessica Spencer, et al. Large-scale identification, mapping, and genotyping of single-nucleotide polymorphisms in the human genome. *Science*, 280(5366):1077–1082, 1998.

[235] Kai Wang, Mingyao Li, and Hakon Hakonarson. Annovar: functional annotation of genetic variants from high-throughput sequencing data. *Nucleic acids research*, 38(16):e164–e164, 2010.

[236] Kai Wang, Mingyao Li, and Hakon Hakonarson. Annovar: functional annotation of genetic variants from high-throughput sequencing data. *Nucleic Acids Research*, 38(16):e164, 2010.

[237] Zhong Wang, Mark Gerstein, and Michael Snyder. Rna-seq: a revolutionary tool for transcriptomics. *Nature reviews genetics*, 10(1):57, 2009.

[238] René L. Warren, Granger G. Sutton, Steven J. M. Jones, and Robert A. Holt. Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501, 2007.

[239] Ann H West and Ann M Stock. Histidine kinases and response regulator proteins in two-component signaling systems. *Trends in Biochemical Sciences*, 26(6):369 – 376, 2001.

[240] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[241] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11):50–60, Nov 2009.

[242] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633 – 652, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[243] Matthew Woitaszek, John M. Dennis, and Taleena R. Sines. Parallel high-resolution climate data analysis using swift. In *Proceedings of the 2011 ACM International Workshop on Many Task Computing on Grids and Supercomputers*, MTAGS '11, pages 5–14, New York, NY, USA, 2011. ACM.

[244] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[245] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3):171–200, Sep 2005.

[246] Jia Yu and Rajkumar Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3-4):217–230, 2006.

[247] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX Login*, 37(4):45–51, 2012.

[248] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[249] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[250] Eleftheria Zeggini, Laura J. Scott, Richa Saxena, Benjamin F. Voight, Jonathan L. Marchini, Tianle Hu, Paul I. W. de Bakker, Gonçalo R. Abecasis, Peter Almgren, Gitte Andersen, Kristin Ardlie, Kristina Bengtsson Boström, Richard N. Bergman, Lori L. Bonnycastle, Knut Borch-Johnsen, Noël P. Burtt, Hong Chen, Peter S. Chines, Mark J. Daly, Parimal Deodhar, Chia-Jen Ding, Alex S. F. Doney, William L. Duren, Katherine S. Elliott, Michael R. Erdos, Timothy M. Frayling, Rachel M. Freathy, Lauren Gianniny, Harald Grallert, Niels Grarup, Christopher J. Groves, Candace Guiducci, Torben Hansen, Christian Herder, Graham A. Hitman, Thomas E. Hughes, Bo Isomaa, Anne U. Jackson, Torben Jørgensen, Augustine Kong, Kari Kubalanza, Finny G. Kuruvilla, Johanna Kuusisto, Claudia Langenberg, Hana Lango, Torsten Lauritzen, Yun Li, Cecilia M. Lindgren, Valeriya Lyssenko, Amanda F. Marvelle, Christa Meisinger, Kristian Midthjell, Karen L. Mohlke, Mario A. Morken, Andrew D. Morris, Narisu Narisu, Peter Nilsson, Katharine R. Owen, Colin N. A. Palmer, Felicity Payne, John R. B. Perry, Elin Pettersen, Carl Platou, Inga Prokopenko, Lu Qi, Li Qin, Nigel W. Rayner, Matthew Rees, Jeffrey J. Roix, Anelli Sandbæk, Beverley Shields, Marketa Sjögren, Valgerdur Steinthorsdottir, Heather M. Stringham, Amy J. Swift, Gudmar Thorleifsson, Unnur Thorsteinsdottir, Nicholas J. Timpson, Tiinamaija Tuomi, Jaakko Tuomilehto, Mark Walker, Richard M. Watanabe, Michael N. Weedon, Cristen J. Willer, Wellcome Trust Case Control Consortium, Thomas Illig, Kristian Hveem,

Frank B. Hu, Markku Laakso, Kari Stefansson, Oluf Pedersen, Nicholas J. Wareham, Inês Barroso, Andrew T. Hattersley, Francis S. Collins, Leif Groop, Mark I. McCarthy, Michael Boehnke, and David Altshuler. Meta-analysis of genome-wide association data and large-scale replication identifies additional susceptibility loci for type 2 diabetes. *Nature Genetics*, 40:638, March 2008.

[251] Yong Zhang, Tao Liu, Clifford A. Meyer, Jérôme Eeckhoute, David S. Johnson, Bradley E. Bernstein, Chad Nusbaum, Richard M. Myers, Myles Brown, Wei Li, and X. Shirley Liu. Model-based analysis of chip-seq (macs). *Genome Biology*, 9(9):R137, Sep 2008.

[252] Daniel Zinn, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Scientific workflow design with data assembly lines. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 14:1–14:10, New York, NY, USA, 2009. ACM.

Appendix A.

Cuneiform Concrete Syntax

```
script       ::= statement+

statement    ::= import
               | query
               | define

query        ::= e ';'

import       ::= 'import' "'...'" ';'

define       ::= let-define
               | fun-define

let-define   ::= 'let' pattern '=' e ';'

fun-define   ::= 'def' id '(' (id ':' type (',' id ':' type)* )? ')'
                 '->' type ( 'in' lang '*{ ... }*' | '{' define* e '}' )

pattern      ::= id ':' type
               | '<' id '=' pattern (',' id '=' pattern)* '>'

lang         ::= 'Awk'
               | 'Bash'
               | 'Elixir'
               | 'Erlang'
               | 'Gnuplot'
               | 'Java'
               | 'Javascript'
               | 'Matlab'
               | 'Octave'
               | 'Perl'
               | 'Python'
               | 'R'
               | 'Racket'

e            ::= var-e
               | app-e
               | str-e
```

```
              | file-e
              | bool-e
              | cond-e
              | list-e
              | hd-e
              | tl-e
              | for-e
              | fold-e
              | record-e
              | proj-e
              | error-e

var-e         ::= id

app-e         ::= id '(' id '=' e (',' id '=' e)* ')'

str-e         ::= '"..."'

file-e        ::= "'...'"

bool-e        ::= 'true'
              | 'false'
              | '(' e '==' e ')'
              | 'not' e
              | '(' e 'and' e ')'
              | '(' e 'or' e ')'
              | 'isnil' e

cond-e        ::= 'if' e 'then' define* e 'else' define* e 'end'

list-e        ::= '[' (e (',' e)*)? ':' type ']'
              | '(' e '>>' e ')'
              | '(' e '+' e ')'

hd-e          ::= 'hd' e 'default' e

tl-e          ::= 'tl' e 'default' e

for-e         ::= 'for' id ':' type '<-' e (',' id ':' type '<-' e)*
                  'do' define* e ':' type 'end'

fold-e        ::= 'fold' id ':' type '=' e ',' id ':' type '<-' e
                   'do' define* e 'end'

record-e      ::= '<' id '=' e (',' id '=' e)* '>'

proj-e        ::= '(' e '|' id ')'

error-e       ::= 'error' '"..."' ':' type
```

```
type         ::= fun-type
               | str-type
               | file-type
               | bool-type
               | list-type
               | record-type

fun-type     ::= 'Fn' '(' (id ':' type (',' id ':' type)* )? ')'
                 '->' type

str-type     ::= 'Str'

file-type    ::= 'File'

bool-type    ::= 'Bool'

list-type    ::= '[' type ']'

record-type  ::= '<' id ':' type (',' id ':' type)* '>'
```

Appendix B.

Advanced Programming Examples

B.1. Ackermann Function

The Ackermann function is an example for a total function that cannot defined using induction-style recursion alone. This means that the function cannot be defined in terms of a function that can be dissected into a base case which is non-recursive and an inductive case that is recursive. The Ackermann function is defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0, n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0, n > 0 \end{cases}$$

The Ackermann function is a function taking two arguments m and n. In addition to general recursion the Ackermann function compares a variable with zero and increments or decrements one its arguments. While recursion, conditions, and string comparison are part of Cuneiform's native language features the arithmetic part, the incrementing and decrementing, must be supplemented from the outside. Here, we use Python as a foreign language.

```
def inc( x : Str ) -> Str {
  def inc( x : Str ) -> <y : Str> in Python *{
    y = int( x )+1
  }*
  ( inc( x = x )|y )
}

def dec( x : Str ) -> Str {
  def dec( x : Str ) -> <y : Str> in Python *{
    y = int( x )-1
  }*
  ( dec( x = x )|y )
}

def ackermann( m : Str, n : Str ) -> Str {
  if( m == 0 )
  then
    inc( x = n )
  else
    if( n == 0 )
```

```
    then
      ackermann(
        m = dec( x = m ),
        n = 1 )
    else
      ackermann(
        m = dec( x = m ),
        n = ackermann( m = m, n = dec( x = n ) ) )
    end
  end
}

ackermann( m = 2, n = 2 );
```

B.2. Quicksort

The Quicksort example uses both aggregation using fold and recursion. The central function in the Quicksort example is the function `sort`. It takes a string list and returns a list of the same size with its elements sorted. First, the function checks whether the argument list is empty or has only one element. In this case, the `sort` function just returns that list. If the list has more than one element `sort` uses the function `pick` to pick the next pivot element. The `pick` function takes a list and returns a random element in this list. Next the `sort` function partitions the input list into three lists using the `partition` function. Herein, the first list contains only elements smaller than the pivot element, the second contains only elements equal to the pivot element, and the third contains only elements larger than the pivot element. Next, the `sort` function calls itself recursively for the first and third partition. Lastly, it appends the three sorted lists and returns the result.

```
def len( lst : [Str] ) -> Str {
  def len( lst : [Str] ) -> <l : Str>
  in Python *{
    l = len( lst )
  }*
  ( len( lst = lst )|l )
}

def pick( lst : [Str] ) -> Str {
  def pick( lst : [Str] ) -> <element : Str>
  in Python *{
    import random
    element = random.choice( lst )
  }*
  ( pick( lst = lst )|element )
}

def le( a : Str, b : Str ) -> Bool {
```

```
  def le( a : Str, b : Str ) -> <isless : Bool>
  in Python *{
      isless = a < b
    }*
  ( le( a = a, b = b )|isless )
}

def partition( lst : [Str], pivot : Str ) ->
  <lo-lst : [Str],
   eq-lst : [Str],
   hi-lst : [Str]> {

  fold acc : <lo-lst : [Str],
              eq-lst : [Str],
              hi-lst : [Str]>
           = <lo-lst = [: Str],
              eq-lst = [: Str],
              hi-lst = [: Str]>,
       s : Str <- lst
  do
    let <lo-lst = lo-lst : [Str],
         eq-lst = eq-lst : [Str],
         hi-lst = hi-lst : [Str]> = acc;

    if( s == pivot )
    then
      <lo-lst = lo-lst,
       eq-lst = ( s >> eq-lst ),
       hi-lst = hi-lst>
    else
      if le( a = s, b = pivot )
      then
        <lo-lst = ( s >> lo-lst ),
         eq-lst = eq-lst,
         hi-lst = hi-lst>
      else
        <lo-lst = lo-lst,
         eq-lst = eq-lst,
         hi-lst = ( s >> hi-lst )>
      end
    end
  end
}

def sort( lst : [Str] ) -> [Str] {

  let l : Str = len( lst = lst );

  if( ( l == 0 ) or ( l == 1 ) )
```

```
  then
    lst
  else

    let pivot : Str =
      pick( lst = lst );

    let <lo-lst = lo-lst : [Str],
         eq-lst = eq-lst : [Str],
         hi-lst = hi-lst : [Str]> =
      partition( lst   = lst,
                 pivot = pivot );

    ( ( sort( lst = lo-lst )+eq-lst )+sort( lst = hi-lst ) )
  end
}

let lst : [Str] =
  [9, 3, 8, 0, 8 : Str];

sort( lst = lst );
```

Appendix C.

Cuneiform Applications

C.1. Variant Calling Using VarScan

```
% VARIANT-CALL
%
% a variant calling workflow
%
%
% Copyright:: 2015-2018 Jörgen Brandt
%
% Licensed under the Apache License, Version 2.0 (the "License");
% you may not use this file except in compliance with the License.
% You may obtain a copy of the License at
%
%    http://www.apache.org/licenses/LICENSE-2.0
%
% Unless required by applicable law or agreed to in writing, software
% distributed under the License is distributed on an "AS IS" BASIS,
% WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
% See the License for the specific language governing permissions and
% limitations under the License.
%
%
% Sample data can be obtained from:
% ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG02025/sequence_read/
%
% The HG38 reference genome can be downloaded from
% http://hgdownload.soe.ucsc.edu/goldenPath/hg38/chromosomes/
%
% An Annovar HG38 database is expected to reside in
% /opt/data/annodb_hg38
%
% In addition to a Cuneiform interpreter the following tools need to be
% installed to run this analysis:
% - FastQC 0.11.4
% - Bowtie2 2.2.6
% - SAMtools 1.2
% - VarScan 2.3.9
% - Annovar
```

```
%
%-------------------------------------------------------------

%% ============================================================
%% Task definitions
%% ============================================================

def split( file : File ) -> <lst : [File]> in Bash *{
  split -l 1280000 $file txt
  lst=txt*
}*

def untar( tar : File ) ->
      <lst : [File]>

in Bash *{
  tar xf $tar
  lst=`tar tf $tar`
}*


def gunzip( gz : File ) ->
      <file : File>

in Bash *{
  file=unzipped_${gz%.gz}
  gzip -c -d $gz > $file
}*


def fastqc( fastq : File ) ->
      <zip : File>

in Bash *{
  fastqc -f fastq --noextract -o ./ $fastq
  zip=`ls *.zip`
}*


def bowtie2-build( fa : File ) ->
      <idx : File>

in Bash *{
  bowtie2-build $fa bt2idx
  idx=idx.tar
  tar cf $idx --remove-files bt2idx.*
}*
```

```
def bowtie2-align( idx : File, fastq1 : File, fastq2 : File ) ->
      <bam : File>

in Bash *{
  tar xf $idx
  bam=alignment.bam
  bowtie2 -D 20 -R 3 -N 0 -L 20 -i S,1,0.50 --no-unal -x bt2idx \
  -1 $fastq1 -2 $fastq2 -S - | samtools view -b - > $bam
  rm bt2idx.*
}*


def samtools-faidx( fa : File ) ->
      <fai : File>

in Bash *{
  samtools faidx $fa
  fai=$fa.fai
}*


def samtools-sort( bam : File ) ->
      <sorted : File>

in Bash *{
  sorted=sorted.bam
  samtools sort -m 2G $bam -o $sorted
}*


def samtools-mpileup( bam : File, fa : File, fai : File ) ->
      <mpileup : File>

in Bash *{
  ln -sf $fai $fa.fai
  mpileup=mpileup.csv
  samtools mpileup -f $fa $bam > $mpileup
}*


def samtools-merge( bam-lst : [File] ) ->
      <merged : File>

{

  def samtools-merge( bams : [File] ) ->
        <merged : File>

  in Bash *{
```

```
    merged=merged.bam
    if [ ${#bams[@]} -eq "1" ]
    then
      merged=$bam
    else
      samtools merge -f $merged ${bams[@]}
    fi
  }*


  if isnil bam-lst
  then
    error "Merge list must not be empty." : <merged : File>
  else
    samtools-merge( bams = bam-lst )
  end
}


def varscan-snp( mpileup : File ) ->
      <vcf : File>

in Bash *{
  vcf=snp.vcf
  varscan mpileup2snp $mpileup --output-vcf > $vcf
}*


def varscan-indel( mpileup : File ) ->
      <vcf : File>

in Bash *{
  vcf=indel.vcf
  varscan mpileup2indel $mpileup --output-vcf > $vcf
}*


def annovar( vcfs : [File], db : File, vsn : Str ) ->
      <fun : File, exonic : File>

in Bash *{
  fun=table.variant_function
  exonic=table.exonic_variant_function
  tar xvf $db
  cat ${vcfs[@]} | \
  convert2annovar.pl -format vcf4 - | \
  annotate_variation.pl -buildver $vsn -geneanno -outfile table - db
}*
```

```
%% ============================================================
%% Input data
%% ============================================================

let hg38-tar : File =
  'hg38/hg38.tar';

let fastq1-lst-gz : [File] =
  ['kgenomes/SRR359188_1.filt.fastq.gz',
   'kgenomes/SRR359195_1.filt.fastq.gz' : File];

let fastq2-lst-gz : [File] =
  ['kgenomes/SRR359188_2.filt.fastq.gz',
   'kgenomes/SRR359195_2.filt.fastq.gz' : File];

let build-vsn : Str =
  "hg38";

let db : File =
  'annovar/hg38db.tar';




%% ============================================================
%% Workflow definition
%% ============================================================

let <lst = fa-lst : [File]> =
  untar( tar = hg38-tar );

let fastq1-lst : [File] =
  for gz : File <- fastq1-lst-gz do
    ( gunzip( gz = gz )|file ) : File
  end;

let fastq2-lst : [File] =
  for gz : File <- fastq2-lst-gz do
    ( gunzip( gz = gz )|file ) : File
  end;

let qc-lst : [File] =
  for fastq : File <- ( fastq1-lst+fastq2-lst ) do
    ( fastqc( fastq = fastq )|zip ) : File
  end;

let mpileup-lst : [File] =
  for fa : File <- fa-lst do
```

```
    let <idx = idx : File> =
      bowtie2-build( fa = fa );

    let <fai = fai : File> =
      samtools-faidx( fa = fa );

    let sorted-lst : [File] =
      for fastq1 : File <- fastq1-lst, fastq2 : File <- fastq2-lst do

        let <lst = split-lst1 : [File]> = split( file = fastq1 );
        let <lst = split-lst2 : [File]> = split( file = fastq2 );

        let bam-lst : [File] =
          for split1 : File <- split-lst1,
              split2 : File <- split-lst2 do

            let <bam = bam : File> =
              bowtie2-align( idx    = idx,
                             fastq1 = split1,
                             fastq2 = split2 );

            ( samtools-sort( bam = bam )|sorted ) : File
          end;

        ( samtools-merge( bam-lst = bam-lst )|merged ) : File

      end;

    let <merged = merged : File> =
      samtools-merge( bam-lst = sorted-lst );

    ( samtools-mpileup( bam = merged,
                        fa  = fa,
                        fai = fai )|mpileup ) : File

  end;

let snp-lst : [File] =
  for mpileup : File <- mpileup-lst do
    ( varscan-snp( mpileup = mpileup )|vcf ) : File
  end;

let indel-lst : [File] =
  for mpileup : File <- mpileup-lst do
    ( varscan-indel( mpileup = mpileup )|vcf ) : File
  end;

let <fun = fun : File, exonic = exonic : File> =
```

```
  annovar( vcfs = ( snp-lst+indel-lst ),
           db   = db,
           vsn  = build-vsn );


%% ============================================================
%% Query
%% ============================================================

<fun    = fun,
 exonic = exonic,
 qc-lst = qc-lst>;
```

C.2. RNA-seq

```
%------------------------------------------------------------------------
% Cuneiform RNA-seq workflow
%
% Copyright 2015-2019 Jörgen Brandt <joergen@cuneiform-lang.org>
%
% Licensed under the Apache License, Version 2.0 (the "License");
% you may not use this file except in compliance with the License.
% You may obtain a copy of the License at
%
%    http://www.apache.org/licenses/LICENSE-2.0
%
% Unless required by applicable law or agreed to in writing, software
% distributed under the License is distributed on an "AS IS" BASIS,
% WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
% See the License for the specific language governing permissions and
% limitations under the License.
%
%------------------------------------------------------------------------

%%======================================================================
%% Function Definitions
%%======================================================================

def samtools-faidx( fa : File ) -> <fai : File>
in Bash *{
  samtools faidx $fa
  fai=$fa.fai
}*

def bowtie-build( fa : File ) -> <idx : File>
in Bash *{
  idx=idx.tar
  bowtie-build $fa idx
  tar cf $idx idx.*
```

```
}*

def tophat-align( fq1      : File,
                  fq2      : File,
                  geneanno : File,
                  fa       : File,
                  idx      : File ) -> <bam : File>
in Bash *{
  tar xf $idx
  ln -sf $fa idx.fa
  tophat -p 8 --bowtie1 -G $geneanno -o thout idx $fq1 $fq2
  bam=thout/accepted_hits.bam
}*

def samtools-index( bam : File ) -> <bai : File>
in Bash *{
  bai=$bam.bai
  samtools index $bam
}*

def samtools-idxstats( bam : File, bai : File ) -> <idxstats : File>
in Bash *{
  ln -sf $bai $bam.bai
  idxstats=idxstats.txt
  samtools idxstats $bam > $idxstats
}*

def cufflinks( bam : File ) -> <transcript : File>
in Bash *{
  cufflinks -p 8 -o clout $bam
  transcript=clout/transcripts.gtf
}*

def cuffmerge( transcriptlst : [File],
               geneanno      : File,
               fa            : File,
               fai           : File ) -> <merged : File>
in Bash *{
  z=genome13581.fa
  ln -s $fa $z
  ln -s $fai ${z}.fai
  printf "%s\n" ${transcriptlst[@]} > assemblies.txt
  cuffmerge -p 8 -g $geneanno -s $z assemblies.txt
  merged=merged_asm/merged.gtf
}*

def cuffdiff( merged : File,
              bam1lst : [File],
              bam2lst : [File],
```

```
              fa       : File,
              fai      : File ) -> <diff : File>
in Bash *{
  b1=‘printf ",%s" ${bam1lst[@]}‘
  b1=${b1:1}

  b2=‘printf ",%s" ${bam2lst[@]}‘
  b2=${b2:1}

  z=genome13582.fa
  ln -s $fa $z
  ln -s $fai ${z}.fai

  cuffdiff -p 8 --no-update-check \
  -o diff_out \
  -b $z \
  -L C1,C2 \
  -u $merged \
  $b1 $b2

  diff=diff.tar
  tar cf $diff diff_out
}*

def cummerbund1( diff : File ) ->
  <csdensity : File,
   scatter : File,
   volcano : File,
   regucalcin_expression : File,
   regucalcin_isoforms : File>
in R *{

  # load libraries
  library(cummeRbund)

  # prepare data directory
  system( paste( "tar xf", diff ) )

  # prepare output directory
  res.out.dir <- "rescb"
  system( paste( "mkdir", res.out.dir ) )


  # create cummerbund database
  cuff_data <- readCufflinks( 'diff_out' )

  # plot distribution of expression levels
  csdensity <- paste( res.out.dir, "csDensity.pdf", sep="/" )
  pdf( csdensity )
```

```
  csDensity( genes( cuff_data) )
  dev.off()

  # compare the expression of each gene in both conditions in a scatter plot
  scatter <- paste( res.out.dir, "csScatter.pdf", sep="/" )
  pdf( scatter )
  csScatter( genes( cuff_data ), 'C1', 'C2' )
  dev.off()

  # compare differentially expressed genes in volcano plot
  volcano <- paste( res.out.dir, "csVolcano.pdf", sep="/" )
  pdf( volcano )
  csVolcano( genes( cuff_data ), 'C1', 'C2'
            # , alpha=0.05, showSignificant=T
  )
  dev.off()

  # define gene of interest regucalcin
  mygene <- getGene( cuff_data, 'regucalcin' )

  # expression levels for gene of interest regucalcin
  regucalcin_expression <-
    paste(
      res.out.dir,
      "regucalcin_expressionBarplot.pdf",
      sep="/" )
  pdf( regucalcin_expression )
  expressionBarplot( mygene )
  dev.off()

  # individual isoform expression levels for gene of interest regucalcin
  regucalcin_isoforms <-
    paste(
      res.out.dir,
      "regucalcin_expressionBarplot_isoforms.pdf",
      sep="/" )
  pdf( regucalcin_isoforms )
  expressionBarplot( isoforms( mygene ) )
  dev.off()
}*

def cummerbund2( diff : File ) ->
  <diff_genes    : File,
   nsig_gene     : Str,
   nsig_isoform  : Str,
   nsig_tss      : Str,
   nsig_cds      : Str,
   nsig_promoter : Str,
   nsig_splicing : Str,
```

```
   nsig_relCDS   : Str>
in R *{

  # load libraries
  library(cummeRbund)

  # prepare data directory
  system( paste( "tar xf", diff ) )

  # prepare output directory
  res.out.dir <- "rescb"
  system( paste( "mkdir", res.out.dir ) )


  # create cummerbund database
  cuff_data <- readCufflinks( 'diff_out' )

  gene_diff_data <- diffData( genes( cuff_data ) )
  sig_gene_data <- subset( gene_diff_data, ( significant == 'yes' ) )
  nsig_gene = nrow( sig_gene_data )

  isoform_diff_data <- diffData( isoforms( cuff_data ), 'C1', 'C2' )
  sig_isoform_data <- subset( isoform_diff_data, ( significant == 'yes' ) )
  nsig_isoform = nrow( sig_isoform_data )

  tss_diff_data <- diffData( TSS( cuff_data ), 'C1', 'C2' )
  sig_tss_data <- subset( tss_diff_data, ( significant == 'yes' ) )
  nsig_tss = nrow( sig_tss_data )

  cds_diff_data <- diffData( CDS( cuff_data ), 'C1', 'C2' )
  sig_cds_data <- subset( cds_diff_data, ( significant == 'yes' ) )
  nsig_cds = nrow( sig_cds_data )

  promoter_diff_data <- distValues( promoters( cuff_data ) )
  sig_promoter_data <- subset( promoter_diff_data, ( significant == 'yes' ) )
  nsig_promoter = nrow( sig_promoter_data )

  splicing_diff_data <- distValues( splicing( cuff_data ) )
  sig_splicing_data <- subset( splicing_diff_data, ( significant == 'yes' ) )
  nsig_splicing = nrow( sig_splicing_data )

  relCDS_diff_data <- distValues(relCDS( cuff_data ) )
  sig_relCDS_data <- subset( relCDS_diff_data, ( significant == 'yes' ) )
  nsig_relCDS = nrow( sig_relCDS_data )

  gene_diff_data <- diffData( genes( cuff_data ) )
  sig_gene_data <- subset( gene_diff_data, ( significant == 'yes' ) )
  diff_genes = 'diff_genes.txt'
  write.table(
```

```
    sig_gene_data,
    diff_genes,
    sep='\t',
    row.names = F,
    col.names = T,
    quote = F )
}*


%%====================================================================
%% Input Data
%%====================================================================

let geneanno : File =
  'BDGP6/Annotation/Genes/genes.gtf';

let fa : File =
  'BDGP6/Sequence/WholeGenomeFasta/genome.fa';

let c1-fq1-lst : [File] =
  ['GSE32038/GSM794483_C1_R1_1.fq',
   'GSE32038/GSM794484_C1_R2_1.fq',
   'GSE32038/GSM794485_C1_R3_1.fq' : File];

let c1-fq2-lst : [File] =
  ['GSE32038/GSM794483_C1_R1_2.fq',
   'GSE32038/GSM794484_C1_R2_2.fq',
   'GSE32038/GSM794485_C1_R3_2.fq' : File];

let c2-fq1-lst : [File] =
  ['GSE32038/GSM794486_C2_R1_1.fq',
   'GSE32038/GSM794487_C2_R2_1.fq',
   'GSE32038/GSM794488_C2_R3_1.fq' : File];

let c2-fq2-lst : [File] =
  ['GSE32038/GSM794486_C2_R1_2.fq',
   'GSE32038/GSM794487_C2_R2_2.fq',
   'GSE32038/GSM794488_C2_R3_2.fq' : File];


%%====================================================================
%% Workflow
%%====================================================================

let <idx = idx : File> =
  bowtie-build( fa = fa );

let c1-bam-lst : [File] =
  for c1-fq1 : File <- c1-fq1-lst,
```

```
      c1-fq2 : File <- c1-fq2-lst do

    ( tophat-align(
        fq1      = c1-fq1,
        fq2      = c1-fq2,
        geneanno = geneanno,
        fa       = fa,
        idx      = idx )|bam ) : File

  end;

let c1-transcript-lst : [File] =
  for c1-bam : File <- c1-bam-lst do
    ( cufflinks( bam = c1-bam )|transcript ) : File
  end;

let c2-bam-lst : [File] =
  for c2-fq1 : File <- c2-fq1-lst,
      c2-fq2 : File <- c2-fq2-lst do

      ( tophat-align(
          fq1      = c2-fq1,
          fq2      = c2-fq2,
          geneanno = geneanno,
          fa       = fa,
          idx      = idx )|bam ) : File

  end;

let c2-transcript-lst : [File] =
  for c2-bam : File <- c2-bam-lst do
    ( cufflinks( bam = c2-bam )|transcript ) : File
  end;

let <fai = fai : File> =
  samtools-faidx( fa = fa );

let <merged = merged : File> =
  cuffmerge(
    transcriptlst  = ( c1-transcript-lst+c2-transcript-lst ),
    geneanno       = geneanno,
    fa             = fa,
    fai            = fai );

let <diff = diff : File> =
  cuffdiff(
    merged  = merged,
    bam1lst = c1-bam-lst,
    bam2lst = c2-bam-lst,
```

```
    fa      = fa,
    fai     = fai );

% steps 9-13
let <csdensity             = csdensity             : File,
     scatter               = scatter               : File,
     volcano               = volcano               : File,
     regucalcin_expression = regucalcin_expression : File,
     regucalcin_isoforms   = regucalcin_isoforms   : File> =
  cummerbund1( diff = diff );

% step 14
let idxstats-lst : [File] =
  for bam : File <- ( c1-bam-lst+c2-bam-lst ) do

    let <bai = bai : File> =
      samtools-index( bam = bam );

    ( samtools-idxstats( bam = bam, bai = bai )|idxstats ) : File

  end;

% we leave out step 15 because cuffcompare can't connect to its update server

% steps 16-18
let <diff_genes    = diff_genes    : File,
     nsig_gene     = nsig_gene     : Str,
     nsig_isoform  = nsig_isoform  : Str,
     nsig_tss      = nsig_tss      : Str,
     nsig_cds      = nsig_cds      : Str,
     nsig_promoter = nsig_promoter : Str,
     nsig_splicing = nsig_splicing : Str,
     nsig_relCDS   = nsig_relCDS   : Str> =
  cummerbund2( diff = diff );


%%====================================================================
%% Query
%%====================================================================

<csdensity             = csdensity,
 scatter               = scatter,
 volcano               = volcano,
 regucalcin_expression = regucalcin_expression,
 regucalcin_isoforms   = regucalcin_isoforms,
 idxstats-lst          = idxstats-lst,
 diff_genes            = diff_genes,
 nsig_gene             = nsig_gene,
 nsig_isoform          = nsig_isoform,
```

```
 nsig_tss              = nsig_tss,
 nsig_cds              = nsig_cds,
 nsig_promoter         = nsig_promoter,
 nsig_splicing         = nsig_splicing,
 nsig_relCDS           = nsig_relCDS>;
```

C.3. ChIP-seq

```
% ChIP-seq
%
% ChIP-seq workflow
%
% Copyright:: 2015-2019 Jörgen Brandt
%
% Licensed under the Apache License, Version 2.0 (the "License");
% you may not use this file except in compliance with the License.
% You may obtain a copy of the License at
%
%    http://www.apache.org/licenses/LICENSE-2.0
%
% Unless required by applicable law or agreed to in writing, software
% distributed under the License is distributed on an "AS IS" BASIS,
% WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
% See the License for the specific language governing permissions and
% limitations under the License.
%
% Input data:
%
% Escherichia coli reference genome
% ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/005/845/
% GCF_000005845.2_ASM584v2/GCF_000005845.2_ASM584v2_genomic.fna.gz
%
% Tag sample
% ftp://ftp-trace.ncbi.nlm.nih.gov/sra/sra-instant/reads/ByStudy/sra/SRP/SRP015/
% SRP015911/SRR576933/SRR576933.sra
%
% Control sample
% ftp://ftp-trace.ncbi.nlm.nih.gov/sra/sra-instant/reads/ByStudy/sra/SRP/SRP015/
% SRP015911/SRR576938/SRR576938.sra
%
%
% Tools and versions:
%
% sra-toolkit 2.8.2-5
% FastQC      0.11.5
% Bowtie      1.2.2
% MACS        1.4.2
% deepTools   3.1.3
% bedtools    2.26.0
```

```
% SAMtools    1.7-1
%
% MACS
% https://github.com/taoliu/MACS/archive/v1.4.2.tar.gz
%
%--------------------------------------------------------------------


%%====================================================================
%% Function definitions
%%====================================================================

def gunzip( gz : File ) -> <file : File>
in Bash *{
  file=unzipped_${gz%.gz}
  gzip -c -d $gz > $file
}*

def fastq-dump( sra : File ) -> <fastq : File>
in Bash *{
  fastq=$sra.fastq
  fastq-dump -Z $sra > $fastq
}*

def fastqc( fastq : File ) -> <zip : File>
in Bash *{
  fastqc -f fastq --noextract -o ./ $fastq
  zip=`ls *.zip`
}*

def bowtie-build( fa : File ) -> <idx : File>
in Bash *{
  idx=idx.tar
  bowtie-build $fa idx
  tar cf $idx idx.*
}*

def bowtie-align( idx : File, fastq : File ) -> <bam : File>
in Bash *{
  bam=$fastq.bam
  tar xf $idx
  bowtie idx -q $fastq -v 2 -m 1 -3 1 -S | samtools view -b - > $bam
}*

def macs( tag : File, ctl : File ) ->
  <peaks   : File,
   summits : File,
   xls     : [File],
   tag_bed : File,
```

```
   ctl_bed : File>
in Bash *{
  macs14 -t $tag \
         -c $ctl \
         --format BAM \
         --gsize 4639675 \
         --name "macs14" \
         --bw 400 \
         --keep-dup 1 \
         --bdg \
         --single-profile \
         --diag

  peaks=macs14_peaks.bed
  summits=macs14_summits.bed
  xls=(macs14_diag.xls macs14_negative_peaks.xls)
  tag_bed=macs14_MACS_bedGraph/treat/macs14_treat_afterfiting_all.bdg.gz
  ctl_bed=macs14_MACS_bedGraph/control/macs14_control_afterfiting_all.bdg.gz
}*

def samtools-sort( bam : File ) -> <sorted : File>
in Bash *{
  sorted=sorted.bam
  samtools sort -m 2G $bam -o $sorted
}*

def samtools-dedup( bam : File ) -> <dedup : File>
in Bash *{
  dedup=dedup.bam
  samtools rmdup -s $bam $dedup
}*

def samtools-index( bam : File ) -> <bai : File>
in Bash *{
  bai=$bam.bai
  samtools index $bam
}*

def samtools-faidx( fa : File ) -> <fai : File>
in Bash *{
  samtools faidx $fa
  fai=$fa.fai
}*

def bamcoverage( bam : File, bai : File ) -> <bedgraph : File>
in Bash *{
  bedgraph=$bam.bedgraph
  ln -sf $bai $bam.bai
  bamCoverage --bam $bam \
```

```
              --outFileName $bedgraph \
              --outFileFormat bedgraph \
              --normalizeUsing RPGC \
              --effectiveGenomeSize 4639675
}*

def restrict-peaks( bed : File ) -> <restricted : File>
in Bash *{
  restricted=$bed.100.bed
  perl -lane '$start=$F[1]+100; $end = $F[2]-100 ; print "$F[0]\t$start\t$end"' \
    $bed > $restricted
}*

def bedtools-getfasta( fa : File, fai : File, bed : File ) ->
  <bed_fa : File>
in Bash *{
  bed_fa=$bed.fa
  ln -sf $fai $fa.fai
  bedtools getfasta -fi $fa -bed $bed -fo $bed_fa
}*

%%====================================================================
%% Input data
%%====================================================================

let ecoli-fa-gz : File = 'GCF_000005845.2_ASM584v2_genomic.fna.gz';
let tag-sra     : File = 'SRR576933';
let ctl-sra     : File = 'SRR576938';


%%====================================================================
%% Workflow definition
%%====================================================================

%% Data preprocessing ------------------------------------------

% decompress E.coli reference genome
let <file = ecoli-fa : File> = gunzip( gz = ecoli-fa-gz );

% convert tag and control samples to FastQ
let <fastq = tag-fastq : File> = fastq-dump( sra = tag-sra );
let <fastq = ctl-fastq : File> = fastq-dump( sra = ctl-sra );

% quality control
let <zip = tag-qc : File> = fastqc( fastq = tag-fastq );
let <zip = ctl-qc : File> = fastqc( fastq = ctl-fastq );

% index reference genome
let <idx = ecoli-idx : File> = bowtie-build( fa = ecoli-fa );
```

```
% read mapping
let <bam = tag-bam : File> =
  bowtie-align( idx   = ecoli-idx,
                fastq = tag-fastq );

let <bam = ctl-bam : File> =
  bowtie-align( idx   = ecoli-idx,
                fastq = ctl-fastq );


%% Peak calling with MACS ----------------------------------------

% call peaks with MACS
let <peaks   = macs-peaks-bed : File,
     summits = macs-summits-bed : File> =
  macs( tag = tag-bam,
        ctl = ctl-bam );

% restrict peaks
let <restricted = restricted-peaks-bed : File> =
  restrict-peaks( bed = macs-peaks-bed );

% extract sequence tag from reference genome
let <fai = ecoli-fai : File> = samtools-faidx( fa = ecoli-fa );

% extract peak DNA sequence
let <bed_fa = restricted-peaks-fa : File> =
  bedtools-getfasta( fa  = ecoli-fa,
                     fai = ecoli-fai,
                     bed = restricted-peaks-bed );


%% coverage information with deepTools -------------------------------

% sort
let <sorted = tag-sorted-bam : File> = samtools-sort( bam = tag-bam );
let <sorted = ctl-sorted-bam : File> = samtools-sort( bam = ctl-bam );

% deduplicate
let <dedup = tag-dedup-bam : File> =
  samtools-dedup( bam = tag-sorted-bam );

let <dedup = ctl-dedup-bam : File> =
  samtools-dedup( bam = ctl-sorted-bam );

% index alignments
let <bai = tag-dedup-bai : File> =
  samtools-index( bam = tag-dedup-bam );
```

```
let <bai = ctl-dedup-bai : File> =
  samtools-index( bam = ctl-dedup-bam );

% coverage information with deepTools
let <bedgraph = tag-deeptools-bedgraph : File> =
  bamcoverage( bam = tag-dedup-bam,
               bai = tag-dedup-bai );

let <bedgraph = ctl-deeptools-bedgraph : File> =
  bamcoverage( bam = ctl-dedup-bam,
               bai = ctl-dedup-bai );




%%====================================================================
%% Query
%%====================================================================

<tag-qc                 = tag-qc,                % quality control
 ctl-qc                 = ctl-qc,

 macs-summits-bed       = macs-summits-bed,      % MACS peak call
 restricted-peaks-bed   = restricted-peaks-bed,
 restricted-peaks-fa    = restricted-peaks-fa,

 tag-deeptools-bedgraph = tag-deeptools-bedgraph, % deepTools coverage
 ctl-deeptools-bedgraph = ctl-deeptools-bedgraph>;
```

C.4. Phylogeny Analysis

```
% PHYLOGENY
%
% phylogeny workflow for CHASE domain in plant proteomes
%
%
% Copyright:: 2016-2019 Jörgen Brandt <joergen@cuneiform-lang.org>
%
% Licensed under the Apache License, Version 2.0 (the "License");
% you may not use this file except in compliance with the License.
% You may obtain a copy of the License at
%
%    http://www.apache.org/licenses/LICENSE-2.0
%
% Unless required by applicable law or agreed to in writing, software
% distributed under the License is distributed on an "AS IS" BASIS,
```

```
% WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
% See the License for the specific language governing permissions and
% limitations under the License.
%
%
% Plant proteome data sources:
% - ftp.gramene.org/pub/gramene/release-60/fasta
% - ftp.jgi-psf.org/pub/JGI_data
%
% In addition to a Cuneiform 3.0.5 interpreter the following tools
% need to be installed:
% - HMMER 3.2.1 (download from http://hmmer.org/)
%   the workflow uses hmmbuild, hmmsearch, esl-translate, and esl-reformat
% - Muscle 3.8.31
% - ALTER 1.3.4
% - PhyML 3.3.3
% - FigTree 1.4.3
%
% Use figtree to view the tree. Asked for a label enter
% "p" for probability. Use seaview to explore the MSA.
%
%------------------------------------------------------------

% default HMMER E-value is 0.01. To find O.tauri the E-value must be no less
% than 0.16.
let evalue : Str = "1e-3";
let maxiters : Str = 512;
let pc : Str = 10;
let model : Str = "LG";
let search : Str = "SPR"; % "NNI";


% Plant aa sequences from Gramene
let gramene-pep-fa-gz-lst : [File] =
  ['gramene/Aegilops_tauschii.Aet_v4.0.pep.all.fa.gz',
   'gramene/Amborella_trichopoda.AMTR1.0.pep.all.fa.gz',
   'gramene/Arabidopsis_halleri.Ahal2.2.pep.all.fa.gz',
   'gramene/Arabidopsis_lyrata.v.1.0.pep.all.fa.gz',
   'gramene/Arabidopsis_thaliana.TAIR10.pep.all.fa.gz',
   'gramene/Beta_vulgaris.RefBeet-1.2.2.pep.all.fa.gz',
   'gramene/Brachypodium_distachyon.Brachypodium_distachyon_v3.0.pep.all.fa.gz',
   'gramene/Brassica_napus.AST_PRJEB5043_v1.pep.all.fa.gz',
   'gramene/Brassica_oleracea.BOL.pep.all.fa.gz',
   'gramene/Brassica_rapa.Brapa_1.0.pep.all.fa.gz',
   'gramene/Chlamydomonas_reinhardtii.Chlamydomonas_reinhardtii_v5.5.pep.all.fa.gz',
   'gramene/Chondrus_crispus.ASM35022v2.pep.all.fa.gz',
   'gramene/Corchorus_capsularis.CCACVL1_1.0.pep.all.fa.gz',
   'gramene/Cucumis_sativus.ASM407v2.pep.all.fa.gz',
   'gramene/Cyanidioschyzon_merolae.ASM9120v1.pep.all.fa.gz',
```

```
   'gramene/Daucus_carota.ASM162521v1.pep.all.fa.gz',
   'gramene/Dioscorea_rotundata.TDr96_F1_Pseudo_Chromosome_v1.0.pep.all.fa.gz',
   'gramene/Galdieria_sulphuraria.ASM34128v1.pep.all.fa.gz',
   'gramene/Glycine_max.Glycine_max_v2.1.pep.all.fa.gz',
   'gramene/Gossypium_raimondii.Graimondii2_0.pep.all.fa.gz',
   'gramene/Helianthus_annuus.HanXRQr1.0.pep.all.fa.gz',
   'gramene/Hordeum_vulgare.IBSC_v2.pep.all.fa.gz',
   'gramene/Leersia_perrieri.Lperr_V1.4.pep.all.fa.gz',
   'gramene/Lupinus_angustifolius.LupAngTanjil_v1.0.pep.all.fa.gz',
   'gramene/Manihot_esculenta.Manihot_esculenta_v6.pep.all.fa.gz',
   'gramene/Medicago_truncatula.MedtrA17_4.0.pep.all.fa.gz',
   'gramene/Musa_acuminata.ASM31385v1.pep.all.fa.gz',
   'gramene/Nicotiana_attenuata.NIATTr2.pep.all.fa.gz',
   'gramene/Oryza_barthii.O.barthii_v1.pep.all.fa.gz',
   'gramene/Oryza_brachyantha.Oryza_brachyantha.v1.4b.pep.all.fa.gz',
   'gramene/Oryza_glaberrima.Oryza_glaberrima_V1.pep.all.fa.gz',
   'gramene/Oryza_glumipatula.Oryza_glumaepatula_v1.5.pep.all.fa.gz',
   'gramene/Oryza_indica.ASM465v1.pep.all.fa.gz',
   'gramene/Oryza_longistaminata.O_longistaminata_v1.0.pep.all.fa.gz',
   'gramene/Oryza_meridionalis.Oryza_meridionalis_v1.3.pep.all.fa.gz',
   'gramene/Oryza_nivara.Oryza_nivara_v1.0.pep.all.fa.gz',
   'gramene/Oryza_punctata.Oryza_punctata_v1.2.pep.all.fa.gz',
   'gramene/Oryza_rufipogon.OR_W1943.pep.all.fa.gz',
   'gramene/Oryza_sativa.IRGSP-1.0.pep.all.fa.gz',
   'gramene/Ostreococcus_lucimarinus.ASM9206v1.pep.all.fa.gz',
   'gramene/Phaseolus_vulgaris.PhaVulg1_0.pep.all.fa.gz',
   'gramene/Physcomitrella_patens.Phypa_V3.pep.all.fa.gz',
   'gramene/Populus_trichocarpa.Pop_tri_v3.pep.all.fa.gz',
   'gramene/Prunus_persica.Prunus_persica_NCBIv2.pep.all.fa.gz',
   'gramene/Selaginella_moellendorffii.v1.0.pep.all.fa.gz',
   'gramene/Setaria_italica.Setaria_italica_v2.0.pep.all.fa.gz',
   'gramene/Solanum_lycopersicum.SL3.0.pep.all.fa.gz',
   'gramene/Solanum_tuberosum.SolTub_3.0.pep.all.fa.gz',
   'gramene/Sorghum_bicolor.Sorghum_bicolor_NCBIv3.pep.all.fa.gz',
   'gramene/Theobroma_cacao.Theobroma_cacao_20110822.pep.all.fa.gz',
   'gramene/Trifolium_pratense.Trpr.pep.all.fa.gz',
   'gramene/Triticum_aestivum.IWGSC.pep.all.fa.gz',
   'gramene/Triticum_dicoccoides.WEWSeq_v.1.0.pep.all.fa.gz',
   'gramene/Triticum_urartu.ASM34745v1.pep.all.fa.gz',
   'gramene/Vigna_angularis.Vigan1.1.pep.all.fa.gz',
   'gramene/Vigna_radiata.Vradiata_ver6.pep.all.fa.gz',
   'gramene/Vitis_vinifera.12X.pep.all.fa.gz',
   'gramene/Zea_mays.B73_RefGen_v4.pep.all.fa.gz' : File];

% Plant aa sequences from JGI
let jgi-pep-fa-gz-lst : [File] =
  ['jgi/agave_deserti_proteins.fa.gz',
   'jgi/agave_tequilana_proteins.fa.gz',
   'jgi/Chlorella_NC64A.all_proteins.fasta.gz',
```

```
    'jgi/Chlvu1_all_proteins_3.fasta.gz',
    'jgi/Dicpu1_all_proteins.fasta.gz',
    'jgi/Emihu1_all_proteins.fasta.gz',
    'jgi/Fracy1_GeneModels_FilteredModels2_aa.fasta.gz',
    'jgi/MicromonasCCMP1545.allModels.aa.fasta.gz',
    'jgi/MicromonasRCC299v3.allModels.proteins.fasta.gz',
    'jgi/Monbr1_all_proteins.fasta.gz',
    'jgi/Naegr1_all_proteins.fasta.gz',
    'jgi/OstreococcusRCC809v2.allModels.proteins.fasta.gz',
    'jgi/Ostta4_GeneModels_AllModels_20111219_aa.fasta.gz',
    'jgi/Phatr1_models_proteins.fasta.gz',
    'jgi/proteins.Auran1_FilteredModels3.fasta.gz',
    'jgi/Pstipitisv2.allModels.proteins.gz',
    'jgi/Thaps3_chromosomes_geneModels_AllModels_20060913_aa.fasta.gz',
    'jgi/Volca1.GeneCatalog_2007_09_13.proteins.fasta.gz',

    % Fungi don't have CHASE domains
    'jgi/CocheC5_1_GeneModels_AllModels_20080320_aa.fasta.gz',
    'jgi/FM1.aa.fasta.gz',
    'jgi/Aspergillus_niger_v3_proteins.fasta.gz',
    'jgi/Batde5_best_proteins.fasta.gz',
    'jgi/allModels.aa.fasta.gz',
    'jgi/Chagl_1_GeneModels_BroadGeneModels_aa.fasta.gz',
    'jgi/Alternaria_brassicicola_proteins.fasta.gz',
    'jgi/Aqu1.pep.fa.gz' : File];

% Microbial genomes from JGI
let jgi-dna-fa-lst : [File] =
  ['jgi-dna/2351348.finished.fsa',
    'jgi-dna/2351356.finished.fsa',
    'jgi-dna/2351359.finished.fsa',
    'jgi-dna/2351362.finished.fsa',
    'jgi-dna/2351472.finished.fsa',
    'jgi-dna/2351473.finished.fsa',
    'jgi-dna/2351474.finished.fsa',
    'jgi-dna/2351476.finished.fsa',
    'jgi-dna/2351477.finished.fsa',
    'jgi-dna/2351479.finished.fsa',
    'jgi-dna/2351482.finished.fsa',
    'jgi-dna/2351488.finished.fsa',
    'jgi-dna/2351493.finished.fsa',
    'jgi-dna/2351519.finished.fsa',
    'jgi-dna/2662178.finished.fsa',
    'jgi-dna/2662179.finished.fsa',
    'jgi-dna/2662180.finished.fsa',
    'jgi-dna/2662182.finished.fsa',
    'jgi-dna/2662184.finished.fsa',
    'jgi-dna/2662185.finished.fsa',
    'jgi-dna/2662186.finished.fsa',
```

```
'jgi-dna/2662187.finished.fsa',
'jgi-dna/2662188.finished.fsa',
'jgi-dna/2662189.finished.fsa',
'jgi-dna/2662192.finished.fsa',
'jgi-dna/2662193.finished.fsa',
'jgi-dna/2662194.finished.fsa',
'jgi-dna/2662195.finished.fsa',
'jgi-dna/2662197.finished.fsa',
'jgi-dna/2662199.finished.fsa',
'jgi-dna/2662200.finished.fsa',
'jgi-dna/2662202.finished.fsa',
'jgi-dna/2662203.finished.fsa',
'jgi-dna/2662205.finished.fsa',
'jgi-dna/2662206.finished.fsa',
'jgi-dna/2662361.finished.fsa',
'jgi-dna/2773013.finished.fsa',
'jgi-dna/2773019.finished.fsa',
'jgi-dna/2773039.finished.fsa',
'jgi-dna/2773040.finished.fsa',
'jgi-dna/2773191.finished.fsa',
'jgi-dna/2773192.finished.fsa',
'jgi-dna/2773254.finished.fsa',
'jgi-dna/2773287.finished.fsa',
'jgi-dna/3435908.finished.fsa',
'jgi-dna/3436081.finished.fsa',
'jgi-dna/3436089.finished.fsa',
'jgi-dna/3436090.finished.fsa',
'jgi-dna/3436091.finished.fsa',
'jgi-dna/3436094.finished.fsa',
'jgi-dna/3436101.finished.fsa',
'jgi-dna/3436109.finished.fsa',
'jgi-dna/3436113.finished.fsa',
'jgi-dna/3436373.finished.fsa',
'jgi-dna/3436384.finished.fsa',
'jgi-dna/3436494.finished.fsa',
'jgi-dna/3436553.finished.fsa',
'jgi-dna/3633862.finished.fsa',
'jgi-dna/3634469.finished.fsa',
'jgi-dna/3634470.finished.fsa',
'jgi-dna/3634472.finished.fsa',
'jgi-dna/3634473.finished.fsa',
'jgi-dna/3634474.finished.fsa',
'jgi-dna/3634475.finished.fsa',
'jgi-dna/3634476.finished.fsa',
'jgi-dna/3634477.finished.fsa',
'jgi-dna/3634478.finished.fsa',
'jgi-dna/3634479.finished.fsa',
'jgi-dna/3634481.finished.fsa',
'jgi-dna/3634482.finished.fsa',
```

```
'jgi-dna/3634483.finished.fsa',
'jgi-dna/3634486.finished.fsa',
'jgi-dna/3634487.finished.fsa',
'jgi-dna/3634488.finished.fsa',
'jgi-dna/3634489.finished.fsa',
'jgi-dna/3634490.finished.fsa',
'jgi-dna/3634491.finished.fsa',
'jgi-dna/3634492.finished.fsa',
'jgi-dna/3634493.finished.fsa',
'jgi-dna/3634494.finished.fsa',
'jgi-dna/3634495.finished.fsa',
'jgi-dna/3634496.finished.fsa',
'jgi-dna/3634497.finished.fsa',
'jgi-dna/3634498.finished.fsa',
'jgi-dna/3634499.finished.fsa',
'jgi-dna/3634500.finished.fsa',
'jgi-dna/3634501.finished.fsa',
'jgi-dna/3634502.finished.fsa',
'jgi-dna/3634503.finished.fsa',
'jgi-dna/3634504.finished.fsa',
'jgi-dna/3634505.finished.fsa',
'jgi-dna/3634510.finished.fsa',
'jgi-dna/3634512.finished.fsa',
'jgi-dna/3634513.finished.fsa',
'jgi-dna/3634543.finished.fsa',
'jgi-dna/3634544.finished.fsa',
'jgi-dna/3634604.finished.fsa',
'jgi-dna/3634605.finished.fsa',
'jgi-dna/3634606.finished.fsa',
'jgi-dna/3634607.finished.fsa',
'jgi-dna/3634642.finished.fsa',
'jgi-dna/3635488.finished.fsa',
'jgi-dna/3635662.finished.fsa',
'jgi-dna/3635680.finished.fsa',
'jgi-dna/3635681.finished.fsa',
'jgi-dna/3635728.finished.fsa',
'jgi-dna/3635729.finished.fsa',
'jgi-dna/3640420.finished.fsa',
'jgi-dna/4000046.finished.fsa',
'jgi-dna/4000097.finished.fsa',
'jgi-dna/4000129.finished.fsa',
'jgi-dna/4000130.finished.fsa',
'jgi-dna/4000135.finished.fsa',
'jgi-dna/4000157.finished.fsa',
'jgi-dna/4000181.finished.fsa',
'jgi-dna/4000182.finished.fsa',
'jgi-dna/4000183.finished.fsa',
'jgi-dna/4000186.finished.fsa',
'jgi-dna/4000187.finished.fsa',
```

```
'jgi-dna/4000203.finished.fsa',
'jgi-dna/4000235.finished.fsa',
'jgi-dna/4000239.finished.fsa',
'jgi-dna/4000241.finished.fsa',
'jgi-dna/4000245.finished.fsa',
'jgi-dna/4000246.finished.fsa',
'jgi-dna/4000263.finished.fsa',
'jgi-dna/4000264.finished.fsa',
'jgi-dna/4000265.finished.fsa',
'jgi-dna/4000266.finished.fsa',
'jgi-dna/4000268.finished.fsa',
'jgi-dna/4000270.finished.fsa',
'jgi-dna/4000271.finished.fsa',
'jgi-dna/4000336.finished.fsa',
'jgi-dna/4000361.finished.fsa',
'jgi-dna/4000362.finished.fsa',
'jgi-dna/4000368.finished.fsa',
'jgi-dna/4000369.finished.fsa',
'jgi-dna/4000370.finished.fsa',
'jgi-dna/4000371.finished.fsa',
'jgi-dna/4000372.finished.fsa',
'jgi-dna/4000373.finished.fsa',
'jgi-dna/4000375.finished.fsa',
'jgi-dna/4000376.finished.fsa',
'jgi-dna/4000377.finished.fsa',
'jgi-dna/4000378.finished.fsa',
'jgi-dna/4000379.finished.fsa',
'jgi-dna/4000380.finished.fsa',
'jgi-dna/4000382.finished.fsa',
'jgi-dna/4000390.finished.fsa',
'jgi-dna/4000415.finished.fsa',
'jgi-dna/4000557.finished.fsa',
'jgi-dna/4000559.finished.fsa',
'jgi-dna/4000602.finished.fsa',
'jgi-dna/4000634.finished.fsa',
'jgi-dna/4000699.finished.fsa',
'jgi-dna/4000715.finished.fsa',
'jgi-dna/4000833.finished.fsa',
'jgi-dna/4000861.finished.fsa',
'jgi-dna/4001067.finished.fsa',
'jgi-dna/4001068.finished.fsa',
'jgi-dna/4001072.finished.fsa',
'jgi-dna/4001073.finished.fsa',
'jgi-dna/4001101.finished.fsa',
'jgi-dna/4001141.finished.fsa',
'jgi-dna/4001178.finished.fsa',
'jgi-dna/4001414.finished.fsa',
'jgi-dna/4001421.finished.fsa',
'jgi-dna/4001423.finished.fsa',
```

```
'jgi-dna/4001425.finished.fsa',
'jgi-dna/4001426.finished.fsa',
'jgi-dna/4001427.finished.fsa',
'jgi-dna/4001428.finished.fsa',
'jgi-dna/4001584.finished.fsa',
'jgi-dna/4001585.finished.fsa',
'jgi-dna/4001606.finished.fsa',
'jgi-dna/4001612.finished.fsa',
'jgi-dna/4001622.finished.fsa',
'jgi-dna/4001624.finished.fsa',
'jgi-dna/4001734.finished.fsa',
'jgi-dna/4001787.finished.fsa',
'jgi-dna/4001789.finished.fsa',
'jgi-dna/4002191.finished.fsa',
'jgi-dna/4002219.finished.fsa',
'jgi-dna/4002277.finished.fsa',
'jgi-dna/4002278.finished.fsa',
'jgi-dna/4002279.finished.fsa',
'jgi-dna/4002280.finished.fsa',
'jgi-dna/4002340.finished.fsa',
'jgi-dna/4002342.finished.fsa',
'jgi-dna/4002390.finished.fsa',
'jgi-dna/4002524.finished.fsa',
'jgi-dna/4002558.finished.fsa',
'jgi-dna/4002584.finished.fsa',
'jgi-dna/4002659.finished.fsa',
'jgi-dna/4002673.finished.fsa',
'jgi-dna/4002674.finished.fsa',
'jgi-dna/4002680.finished.fsa',
'jgi-dna/4002681.finished.fsa',
'jgi-dna/4002686.finished.fsa',
'jgi-dna/4002719.finished.fsa',
'jgi-dna/4002720.finished.fsa',
'jgi-dna/4002721.finished.fsa',
'jgi-dna/4002722.finished.fsa',
'jgi-dna/4002725.finished.fsa',
'jgi-dna/4002730.finished.fsa',
'jgi-dna/4002732.finished.fsa',
'jgi-dna/4002733.finished.fsa',
'jgi-dna/4002758.finished.fsa',
'jgi-dna/4002759.finished.fsa',
'jgi-dna/4002760.finished.fsa',
'jgi-dna/4002761.finished.fsa',
'jgi-dna/4002762.finished.fsa',
'jgi-dna/4002766.finished.fsa',
'jgi-dna/4002768.finished.fsa',
'jgi-dna/4002885.finished.fsa',
'jgi-dna/4002915.finished.fsa',
'jgi-dna/4002917.finished.fsa',
```

```
'jgi-dna/4002919.finished.fsa',
'jgi-dna/4002939.finished.fsa',
'jgi-dna/4002943.finished.fsa',
'jgi-dna/4002947.finished.fsa',
'jgi-dna/4002948.finished.fsa',
'jgi-dna/4003005.finished.fsa',
'jgi-dna/4003027.finished.fsa',
'jgi-dna/4003028.finished.fsa',
'jgi-dna/4003030.finished.fsa',
'jgi-dna/4003071.finished.fsa',
'jgi-dna/4003073.finished.fsa',
'jgi-dna/4003074.finished.fsa',
'jgi-dna/4003075.finished.fsa',
'jgi-dna/4003083.finished.fsa',
'jgi-dna/4003084.finished.fsa',
'jgi-dna/4003208.finished.fsa',
'jgi-dna/4003209.finished.fsa',
'jgi-dna/4003319.finished.fsa',
'jgi-dna/4003779.finished.fsa',
'jgi-dna/4003781.finished.fsa',
'jgi-dna/4003782.finished.fsa',
'jgi-dna/4003783.finished.fsa',
'jgi-dna/4003784.finished.fsa',
'jgi-dna/4003785.finished.fsa',
'jgi-dna/4003799.finished.fsa',
'jgi-dna/4003800.finished.fsa',
'jgi-dna/4003801.finished.fsa',
'jgi-dna/4003939.finished.fsa',
'jgi-dna/4003959.finished.fsa',
'jgi-dna/4004019.finished.fsa',
'jgi-dna/4004020.finished.fsa',
'jgi-dna/4004021.finished.fsa',
'jgi-dna/4005180.finished.fsa',
'jgi-dna/4005181.finished.fsa',
'jgi-dna/4005359.finished.fsa',
'jgi-dna/4023460.finished.fsa',
'jgi-dna/4023462.finished.fsa',
'jgi-dna/4023464.finished.fsa',
'jgi-dna/4023468.finished.fsa',
'jgi-dna/4023470.finished.fsa',
'jgi-dna/4023472.finished.fsa',
'jgi-dna/4023685.finished.fsa',
'jgi-dna/4023905.finished.fsa',
'jgi-dna/4024116.finished.fsa',
'jgi-dna/4024122.finished.fsa',
'jgi-dna/4024126.finished.fsa',
'jgi-dna/4024128.finished.fsa',
'jgi-dna/4024132.finished.fsa',
'jgi-dna/4024134.finished.fsa',
```

```
'jgi-dna/4024136.finished.fsa',
'jgi-dna/4024143.finished.fsa',
'jgi-dna/4024147.finished.fsa',
'jgi-dna/4024149.finished.fsa',
'jgi-dna/4024151.finished.fsa',
'jgi-dna/4024153.finished.fsa',
'jgi-dna/4024171.finished.fsa',
'jgi-dna/4024173.finished.fsa',
'jgi-dna/4024175.finished.fsa',
'jgi-dna/4024181.finished.fsa',
'jgi-dna/4024183.finished.fsa',
'jgi-dna/4042873.finished.fsa',
'jgi-dna/4042883.finished.fsa',
'jgi-dna/4042952.finished.fsa',
'jgi-dna/4042953.finished.fsa',
'jgi-dna/4042956.finished.fsa',
'jgi-dna/4042958.finished.fsa',
'jgi-dna/4043073.finished.fsa',
'jgi-dna/4043133.finished.fsa',
'jgi-dna/4043135.finished.fsa',
'jgi-dna/4044004.finished.fsa',
'jgi-dna/4044106.finished.fsa',
'jgi-dna/4075091.finished.fsa',
'jgi-dna/4075103.finished.fsa',
'jgi-dna/4082343.finished.fsa',
'jgi-dna/4082369.finished.fsa',
'jgi-dna/4082373.finished.fsa',
'jgi-dna/4082375.finished.fsa',
'jgi-dna/4082379.finished.fsa',
'jgi-dna/4082381.finished.fsa',
'jgi-dna/4082383.finished.fsa',
'jgi-dna/4082385.finished.fsa',
'jgi-dna/4082401.finished.fsa',
'jgi-dna/4082513.finished.fsa',
'jgi-dna/4082525.finished.fsa',
'jgi-dna/4082717.finished.fsa',
'jgi-dna/4082733.finished.fsa',
'jgi-dna/4082737.finished.fsa',
'jgi-dna/4082741.finished.fsa',
'jgi-dna/4082745.finished.fsa',
'jgi-dna/4082753.finished.fsa',
'jgi-dna/4082757.finished.fsa',
'jgi-dna/4082761.finished.fsa',
'jgi-dna/4082765.finished.fsa',
'jgi-dna/4082769.finished.fsa',
'jgi-dna/4082773.finished.fsa',
'jgi-dna/4082785.finished.fsa',
'jgi-dna/4082789.finished.fsa',
'jgi-dna/4082793.finished.fsa',
```

```
'jgi-dna/4082797.finished.fsa',
'jgi-dna/4082801.finished.fsa',
'jgi-dna/4082850.finished.fsa',
'jgi-dna/4082854.finished.fsa',
'jgi-dna/4082943.finished.fsa',
'jgi-dna/4082953.finished.fsa',
'jgi-dna/4082961.finished.fsa',
'jgi-dna/4082966.finished.fsa',
'jgi-dna/4082970.finished.fsa',
'jgi-dna/4082974.finished.fsa',
'jgi-dna/4082980.finished.fsa',
'jgi-dna/4082984.finished.fsa',
'jgi-dna/4082988.finished.fsa',
'jgi-dna/4082992.finished.fsa',
'jgi-dna/4082996.finished.fsa',
'jgi-dna/4083000.finished.fsa',
'jgi-dna/4083004.finished.fsa',
'jgi-dna/4083008.finished.fsa',
'jgi-dna/4083012.finished.fsa',
'jgi-dna/4083016.finished.fsa',
'jgi-dna/4083028.finished.fsa',
'jgi-dna/4083041.finished.fsa',
'jgi-dna/4083050.finished.fsa',
'jgi-dna/4083212.finished.fsa',
'jgi-dna/4083220.finished.fsa',
'jgi-dna/4083224.finished.fsa',
'jgi-dna/4083228.finished.fsa',
'jgi-dna/4083242.finished.fsa',
'jgi-dna/4083246.finished.fsa',
'jgi-dna/4083250.finished.fsa',
'jgi-dna/4083258.finished.fsa',
'jgi-dna/4083268.finished.fsa',
'jgi-dna/4083272.finished.fsa',
'jgi-dna/4083288.finished.fsa',
'jgi-dna/4083292.finished.fsa',
'jgi-dna/4083296.finished.fsa',
'jgi-dna/4083304.finished.fsa',
'jgi-dna/4083308.finished.fsa',
'jgi-dna/4083312.finished.fsa',
'jgi-dna/4083320.finished.fsa',
'jgi-dna/4083324.finished.fsa',
'jgi-dna/4083328.finished.fsa',
'jgi-dna/4083332.finished.fsa',
'jgi-dna/4083340.finished.fsa',
'jgi-dna/4083512.finished.fsa',
'jgi-dna/4083620.finished.fsa',
'jgi-dna/4083788.finished.fsa',
'jgi-dna/4083792.finished.fsa',
'jgi-dna/4083800.finished.fsa',
```

```
'jgi-dna/4083905.finished.fsa',
'jgi-dna/4083909.finished.fsa',
'jgi-dna/4084069.finished.fsa',
'jgi-dna/4084073.finished.fsa',
'jgi-dna/4084098.finished.fsa',
'jgi-dna/4084143.finished.fsa',
'jgi-dna/4084198.finished.fsa',
'jgi-dna/4084204.finished.fsa',
'jgi-dna/4084270.finished.fsa',
'jgi-dna/4084274.finished.fsa',
'jgi-dna/4084296.finished.fsa',
'jgi-dna/4084304.finished.fsa',
'jgi-dna/4084488.finished.fsa',
'jgi-dna/4084538.finished.fsa',
'jgi-dna/4084589.finished.fsa',
'jgi-dna/4084710.finished.fsa',
'jgi-dna/4084990.finished.fsa',
'jgi-dna/4084991.finished.fsa',
'jgi-dna/4085007.finished.fsa',
'jgi-dna/4085008.finished.fsa',
'jgi-dna/4085026.finished.fsa',
'jgi-dna/4085029.finished.fsa',
'jgi-dna/4085030.finished.fsa',
'jgi-dna/4085034.finished.fsa',
'jgi-dna/4085035.finished.fsa',
'jgi-dna/4085036.finished.fsa',
'jgi-dna/4085037.finished.fsa',
'jgi-dna/4085038.finished.fsa',
'jgi-dna/4085041.finished.fsa',
'jgi-dna/4085042.finished.fsa',
'jgi-dna/4085044.finished.fsa',
'jgi-dna/4085075.finished.fsa',
'jgi-dna/4085125.finished.fsa',
'jgi-dna/4085179.finished.fsa',
'jgi-dna/4085235.finished.fsa',
'jgi-dna/4085236.finished.fsa',
'jgi-dna/4085237.finished.fsa',
'jgi-dna/4085254.finished.fsa',
'jgi-dna/4085263.finished.fsa',
'jgi-dna/4085264.finished.fsa',
'jgi-dna/4085283.finished.fsa',
'jgi-dna/4085294.finished.fsa',
'jgi-dna/4085314.finished.fsa',
'jgi-dna/4085515.finished.fsa',
'jgi-dna/4085516.finished.fsa',
'jgi-dna/4085555.finished.fsa',
'jgi-dna/4085559.finished.fsa',
'jgi-dna/4085568.finished.fsa',
'jgi-dna/4085576.finished.fsa',
```

```
'jgi-dna/4085666.finished.fsa',
'jgi-dna/4085667.finished.fsa',
'jgi-dna/4085668.finished.fsa',
'jgi-dna/4085694.finished.fsa',
'jgi-dna/4085699.finished.fsa',
'jgi-dna/4085704.finished.fsa',
'jgi-dna/4085705.finished.fsa',
'jgi-dna/4085717.finished.fsa',
'jgi-dna/4085721.finished.fsa',
'jgi-dna/4085722.finished.fsa',
'jgi-dna/4085724.finished.fsa',
'jgi-dna/4085738.finished.fsa',
'jgi-dna/4085750.finished.fsa',
'jgi-dna/4085751.finished.fsa',
'jgi-dna/4085752.finished.fsa',
'jgi-dna/4085823.finished.fsa',
'jgi-dna/4085824.finished.fsa',
'jgi-dna/4085842.finished.fsa',
'jgi-dna/4086078.finished.fsa',
'jgi-dna/4086087.finished.fsa',
'jgi-dna/4086088.finished.fsa',
'jgi-dna/4086101.finished.fsa',
'jgi-dna/4086172.finished.fsa',
'jgi-dna/4086173.finished.fsa',
'jgi-dna/4086180.finished.fsa',
'jgi-dna/4086181.finished.fsa',
'jgi-dna/4086190.finished.fsa',
'jgi-dna/4086216.finished.fsa',
'jgi-dna/4086217.finished.fsa',
'jgi-dna/4086221.finished.fsa',
'jgi-dna/4086226.finished.fsa',
'jgi-dna/4086227.finished.fsa',
'jgi-dna/4086228.finished.fsa',
'jgi-dna/4086229.finished.fsa',
'jgi-dna/4086230.finished.fsa',
'jgi-dna/4086259.finished.fsa',
'jgi-dna/4086261.finished.fsa',
'jgi-dna/4086262.finished.fsa',
'jgi-dna/4086293.finished.fsa',
'jgi-dna/4086324.finished.fsa',
'jgi-dna/4086336.finished.fsa',
'jgi-dna/4086390.finished.fsa',
'jgi-dna/4086435.finished.fsa',
'jgi-dna/4086469.finished.fsa',
'jgi-dna/4086483.finished.fsa',
'jgi-dna/4086500.finished.fsa',
'jgi-dna/4086508.finished.fsa',
'jgi-dna/4086511.finished.fsa',
'jgi-dna/4086585.finished.fsa',
```

```
   'jgi-dna/4086743.finished.fsa',
   'jgi-dna/4086746.finished.fsa',
   'jgi-dna/4086749.finished.fsa',
   'jgi-dna/4086827.finished.fsa',
   'jgi-dna/4086836.finished.fsa',
   'jgi-dna/4086840.finished.fsa',
   'jgi-dna/4086861.finished.fsa',
   'jgi-dna/4086885.finished.fsa',
   'jgi-dna/4087032.finished.fsa',
   'jgi-dna/4087033.finished.fsa',
   'jgi-dna/4087034.finished.fsa',
   'jgi-dna/4087036.finished.fsa',
   'jgi-dna/4087133.finished.fsa',
   'jgi-dna/4087334.finished.fsa',
   'jgi-dna/4087335.finished.fsa',
   'jgi-dna/4087337.finished.fsa',
   'jgi-dna/4087339.finished.fsa',
   'jgi-dna/4087341.finished.fsa',
   'jgi-dna/4087345.finished.fsa',
   'jgi-dna/4087350.finished.fsa',
   'jgi-dna/4087355.finished.fsa',
   'jgi-dna/4087361.finished.fsa',
   'jgi-dna/4087603.finished.fsa',
   'jgi-dna/4087653.finished.fsa',
   'jgi-dna/4087656.finished.fsa',
   'jgi-dna/4087657.finished.fsa',
   'jgi-dna/4087668.finished.fsa',
   'jgi-dna/4088690.finished.fsa',
   'jgi-dna/4088693.finished.fsa',
   'jgi-dna/4088792.finished.fsa',
   'jgi-dna/4088881.finished.fsa',
   'jgi-dna/4089210.finished.fsa',
   'jgi-dna/4089449.finished.fsa',
   'jgi-dna/4089466.finished.fsa',
   'jgi-dna/4089467.finished.fsa',
   'jgi-dna/4090073.finished.fsa',
   'jgi-dna/4091580.finished.fsa' : File];


%%=====================================================================
%% Task definitions
%%=====================================================================

def beautify-tag( fa : File ) -> <result : File> in Awk *{
  /^>fgeneshNG/      { printf ">Arabidopsis_lyrata%u\n", ++araly; next }
  /^>fgenesh2/       { printf ">Arabidopsis_lyrata%u\n", ++araly; next }
  /^>fgeneshHS/      { printf ">Naegleria_gruberi%u\n", ++naegr; next }
  /^>estExt/         { printf ">Naegleria_gruberi%u\n", ++naegr; next }
```

```
/^>.*Ahal/         { printf ">Arabidopsis_halleri%u\n", ++araha; next }
/^>.*TAIR10/       { printf ">Arabidopsis_thaliana%u\n", ++arath; next }
/^>.*Sorghum_bicolor/ { printf ">Sorghum_bicolor%u\n", ++sorbi; next}
/^>.*Pop_tri/      { printf ">Populus_trichocarpa%u\n", ++poptr; next }
/^>.*Brachypodium_distachyon/ { printf ">Brachypodium_distachyon%u\n", ++bradi; next }
/^>.*LupAng/       { printf ">Lupinus_angustifolius%u\n", ++lupan; next }
/^>.*NIATT/        { printf ">Nicotiana_attenuata%u\n", ++nicat; next }
/^>.*Medtr/        { printf ">Medicago_truncatula%u\n", ++medtr; next }

/^>AET/            { printf ">Aegilops_tauschii%u\n", ++aet; next }
/^>BGIOS/          { printf ">Oryza_indica%u\n", ++bgios; next }
/^>Bo/             { printf ">Brassica_oleracea%u\n", ++bo; next }
/^>Bra/            { printf ">Brassica_rapa%u\n", ++bra; next }
/^>CD/             { printf ">Brassica_napus%u\n", ++cd; next }
/^>Dr/             { printf ">Dioscorea_rotundata%u\n", ++dr; next }
/^>EFJ/            { printf ">Selaginella_moellendorffii%u\n", ++efj; next}
/^>EO/             { printf ">Theobromba_cacao%u\n", ++eo; next}
/^>ERN/            { printf ">Amborella_trichopoda%u\n", ++ern; next}
/^>ESW/            { printf ">Phaseolus_vulgaris%u\n", ++esw; next}
/^>GSMUA/          { printf ">Musa_acuminata%u\n", ++gsmua; next}
/^>HORVU/          { printf ">Hordeum_vulgare%u\n", ++horvu; next }
/^>KGN/            { printf ">Cucumis_sativus%u\n", ++kgn; next }
/^>KJB/            { printf ">Gossypium_raimondii%u\n", ++kjb; next }
/^>KM/             { printf ">Beta_vulgaris%u\n", ++kms; next }
/^>KN/             { printf ">Oryza_longistaminata%u\n", ++kn; next }
/^>KOM/            { printf ">Vigna_angularis%u\n", ++kom; next }
/^>KQ/             { printf ">Setaria_italica%u\n", ++kq; next }
/^>KRH/            { printf ">Glycine_max%u\n", ++krh; next }
/^>KZ/             { printf ">Daucus_carota%u\n", ++kz; next }
/^>LPERR/          { printf ">Leersia_perrieri%u\n", ++lperr; next }
/^>OAY/            { printf ">Manihot_esculenta%u\n", ++oay; next }
/^>OB/             { printf ">Oryza_brachyantha%u\n", ++ob; next }
/^>OGLUM/          { printf ">Oryza_glumipatula%u\n", ++oglum; next }
/^>OMERI/          { printf ">Oryza_meridionalis%u\n", ++omeri; next }
/^>OM/             { printf ">Corchorus_capsularis%u\n", ++om; next }
/^>ONIVA/          { printf ">Oryza_nivara%u\n", ++oniva; next }
/^>ONI/            { printf ">Prunus_persica%u\n", ++oni; next }
/^>OPUNC/          { printf ">Oryza_punctata%u\n", ++opunc; next }
/^>ORGLA/          { printf ">Oryza_glaberrima%u\n", ++orgla; next }
/^>ORUFI/          { printf ">Oryza_rufipogon%u\n", ++orufi; next }
/^>OTG/            { printf ">Helianthus_annuus%u\n", ++otg; next }
/^>Os/             { printf ">Oryza_sativa%u\n", ++os; next }
/^>PGSC/           { printf ">Solanum_tuberosum%u\n", ++pgsc; next }
/^>PNW/            { printf ">Chlamydomonas_reinhardtii%u\n", ++pgsc; next }
/^>Pp/             { printf ">Physcomitrella_patens%u\n", ++pp; next }
/^>Solyc/          { printf ">Solanum_lycopersicum%u\n", ++solyc; next }
/^>Traes/          { printf ">Triticum_aestivum%u\n", ++traes; next }
/^>TRIDC/          { printf ">Triticum_dicoccoides%u\n", ++tridc; next }
/^>TRIUR/          { printf ">Triticum_urartu%u\n", ++triur; next }
```

```
  /^>Tp/             { printf ">Trifolium_pratense%u\n", ++tp; next }
  /^>vigra/          { printf ">Vigna_radiata%u\n", ++vigra; next }
  /^>VIT/            { printf ">Vitis_vinifera%u\n", ++vit; next }
  /^>Zm/             { printf ">Zea_mays%u\n", ++zm; next }

  /^>jgi\|Agade1\|/  { printf ">Agave_deserti%u\n", ++agade; next }
  /^>jgi\|Agate1\|/  { printf ">Agave_tequilana%u\n", ++agate; next }
  /^>jgi\|Auran1\|/  { printf ">Aureococcus_anophagefferens%u\n", ++auran; next }
  /^>jgi\|Chlvu1\|/  { printf ">Chlorella_vulgaris%u\n", ++chlvu; next }
  /^>jgi\|Cioin2\|/  { printf ">Ciona_intestinalis%u\n", ++cioin; next }
  /^>jgi\|Dicpu1\|/  { printf ">Dictyostelium_purpureum%u\n", ++dicpu; next }
  /^>jgi\|MicpuC2\|/ { printf ">Micromonas_pusilla_CCMP1545%u\n", ++micpuc; next }
  /^>jgi\|Ostta4\|/  { printf ">Ostreococcus_tauri%u\n", ++ostta; next }
  /^>jgi\|Phatr1\|/  { printf ">Phaeodactylum_tricornutum%u\n", ++phatr; next }
  /^>jgi\|Thaps3\|/  { printf ">Thalassiosira_pseudonana%u\n", ++thaps; next }
  /^>jgi\|Volca1\|/  { printf ">Volvox_carteri%u\n", ++volca; next }

  /^>.*\[organism=/  { match( $0, /\[organism=([A-Za-z0-9\.\-_ ]*)/, a )
                       gsub( " ", "_", a[1] )
                       printf ">%s%u\n", a[1], ++microbial
                       next }

                     { print }
}*

def gunzip( gz : File ) -> <out : File>
in Bash *{
  out=unzipped_${gz%.gz}
  gzip -c -d $gz > $out
}*

def cat( lst : [File] ) -> <out : File>
in Bash *{
  out=out.txt
  cat ${lst[@]} > $out
}*

def extract-orf( fa : File ) -> <orf : File>
in Bash *{
  orf=orf.fa
  awk '/^\/\/$/ { next } /^complete sequence$/ { next } { print }' $fa | esl-translate -c 11 - > $orf
}*

def hmmbuild( sto : File ) -> <hmm : File>
in Bash *{
  hmm=${sto%.sto}.hmm
  hmmbuild --cpu 1 --amino $hmm $sto
}*
```

```
def sto2fa( sto : File ) -> <fa : File>
in Bash *{
  fa=${sto%.sto}.fa
  if [ -s $sto ]
  then
    esl-reformat fasta $sto > $fa
  else
    touch $fa
  fi
}*

def hmmsearch( hmm : File, fa : File, evalue : Str ) -> <hits : File>
in Bash *{
  hits=hits_${fa%.fa}.sto
  hmmsearch --cpu 1 --incE $evalue -A $hits $hmm $fa
}*

def muscle( fa : File, maxiters : Str ) -> <msa : File, log : File>
in Bash *{
  msa=msa.fa
  log=muscle.log
  muscle -maxiters $maxiters -in $fa -out $msa -log $log
}*

def alter( msa : File ) -> <phy : File>
in Bash *{
  phy=msa.phy
  alter-sequence-alignment \
    -i $msa -if FASTA -io Linux -ip MUSCLE \
    -o $phy -of PHYLIP -oo Linux -op PhyML
}*

def phyml( phy : File, model : Str, search : Str ) -> <tree : File, stats : File>
in Bash *{
  tree=${phy}_phyml_tree.txt
  stats=${phy}_phyml_stats.txt
  phyml -d aa --no_memory_check -m $model -s $search -i $phy
}*

def figtree( tree : File ) -> <png : File>
in Bash *{
  png=tree.png
  figtree -graphic PNG -width 1280 -height 1280 $tree $png
}*


%%=====================================================================
%% Input data
%%=====================================================================
```

```
% CHASE seed alignment
let chase-seed-sto : File = 'pfam/PF03924_seed.txt';



%%====================================================================
%% Workflow definition
%%====================================================================

let pep-fa-gz-lst : [File] =
  ( gramene-pep-fa-gz-lst+jgi-pep-fa-gz-lst );

let pep-fa-lst : [File] =
  for fa-gz : File <- pep-fa-gz-lst do
    ( gunzip( gz = fa-gz )|out ) : File
  end;

let jgi-orf-fa-lst : [File] =
  for fa : File <- jgi-dna-fa-lst do
    ( extract-orf( fa = fa )|orf ) : File
  end;

let fa-lst : [File] =
  ( pep-fa-lst+jgi-orf-fa-lst );

let <hmm = hmm : File> =
  hmmbuild( sto = chase-seed-sto );

let hits-fa-lst : [File] =
  for fa : File <- fa-lst do

    let <hits = chase-full-sto : File> =
      hmmsearch( hmm    = hmm,
                 fa     = fa,
                 evalue = evalue );

    let <fa = chase-full-fa : File> =
      sto2fa( sto = chase-full-sto );

    chase-full-fa : File

  end;

let <out = hits-fa : File> =
  cat( lst = hits-fa-lst );

let <result = beautiful-hits-fa : File> =
  beautify-tag( fa = hits-fa );
```

```
let <msa = msa : File, log = muscle-log : File> =
  muscle( fa       = beautiful-hits-fa,
          maxiters = maxiters );

let <phy = phy : File> =
  alter( msa = msa );

let <tree = tree : File, stats = phyml-stats : File> =
  phyml( phy = phy, model = model, search = search );

let <png = png : File> =
  figtree( tree = tree );

%%====================================================================
%% Query
%%====================================================================

<tree = tree, png = png>;
```

C.5. Distributed *K*-means

```
%%============================================================
%% Utility functions
%%============================================================

def reverse-file-lst( lst : [File] ) -> [File] {
  fold acc : [File] = [: File], x : File <- lst do
    (x >> acc)
  end
}

def last-file( lst : [File], default : File ) -> File {
  fold acc : File = default, x : File <- lst do
    x
  end
}

def first-file( lst : [File], default : File ) -> File {
  last-file(
    lst     = reverse-file-lst( lst = lst ),
    default = default )
}

%%============================================================
%% Foreign functions
%%============================================================

def gen-random-sample( k : Str, n : Str ) ->
```

```
  <cluster-lst : [File]>
in Racket *{

  (require (only-in k-means/gen
                    gen-random-sample))

  (define cluster-lst
    (build-list (string->number k)
                (lambda (i) (number->string i))))

  (define c-lst
    (gen-random-sample (string->number k)
                       (string->number n)))

  (define (proc f c)
    (call-with-output-file f
      (lambda (out) (write c out))))

  (for-each proc cluster-lst c-lst)
}*

def shuffle-split( cluster-lst : [File], n : Str ) ->
  <split-lst : [File]>
in Racket *{

  (require (only-in racket/list
                    shuffle)

           (only-in k-means
                    split))

  (define (in-proc f)
    (call-with-input-file f
      (lambda (in) (read in))))

  (define (out-proc f s)
    (call-with-output-file f
      (lambda (out) (write s out))))

  (define s-lst
    (split
      (shuffle (apply append (map in-proc cluster-lst)))
      (string->number n)))

  (define split-lst
    (build-list (string->number n)
                (lambda (i) (number->string i))))

  (for-each out-proc split-lst s-lst)
```

```
}*

def init-cc-lst( k : Str ) -> <cc-lst : File>
in Racket *{

  (require (only-in k-means/gen
                    gen-init-cc-lst))

  (define l
    (gen-init-cc-lst (string->number k)))

  (define cc-lst
    "cc-lst")

  (call-with-output-file cc-lst
                         (lambda (out) (write l out)))
}*

def has-converged( a : File, b : File ) -> <p : Bool>
in Racket *{

  (define (proc in)
    (read in))

  (define cc-lst-a
    (call-with-input-file a proc))

  (define cc-lst-b
    (call-with-input-file b proc))

  (define p
    (equal? cc-lst-a cc-lst-b))
}*

def weighted-mean( pair-lst : [File] ) ->
  <mean-cc-lst : File>
in Racket *{

  (require (only-in k-means
                    weighted-mean))

  (define (in-proc f)
    (call-with-input-file f
      (lambda (in) (read in))))

  (define cluster-pair-lst
    (map in-proc pair-lst))

  (displayln cluster-pair-lst)
```

```
  (define m
    (weighted-mean cluster-pair-lst))

  (define mean-cc-lst
    "mean-cc-lst")

  (call-with-output-file mean-cc-lst
                         (lambda (out) (write m out)))

}*

def step-split( split : File, cc-lst : File ) ->
  <cluster-pair : File>
in Racket *{

  (require (only-in k-means
                    partition
                    cluster-center))

  (define s
    (call-with-input-file split (lambda (in) (read in))))

  (define c0
    (call-with-input-file cc-lst (lambda (in) (read in))))

  (define partition-lst
    (partition s c0))

  (define c1
    (for/list ([partition partition-lst])
          (cluster-center partition)))

  (define p
    (cons c1 (length s)))

  (define cluster-pair
    "cluster-pair")

  (call-with-output-file
    cluster-pair
    (lambda (out) (write p out)))

}*

%% k-means algorithm ------------------------------------

def step( split-lst : [File], cc-lst : File ) -> File {
```

```
  let pair-lst : [File] =
    for split : File <- split-lst do

      let <cluster-pair = cluster-pair : File> =
        step-split( split  = split,
                    cc-lst = cc-lst );

      cluster-pair : File
    end;

  let <mean-cc-lst = new-cc-lst : File> =
    weighted-mean( pair-lst = pair-lst );

  new-cc-lst
}

def run-k-means( split-lst : [File], history : [File] ) ->
  [File] {

    let cc-lst : File =
      first-file(
        lst     = history,
        default = error "history must not be empty" : File );

    let new-cc-lst : File =
      step( split-lst = split-lst,
            cc-lst    = cc-lst );

    let <p = converged : Bool> =
      has-converged( a = cc-lst,
                     b = new-cc-lst );

    if
      converged
    then
      reverse-file-lst( lst = history )
    else
      run-k-means( split-lst = split-lst,
                   history   = ( new-cc-lst >> history ) )
    end
}

%% plotting ------------------------------------------------

def plot-random-sample( cluster-lst : [File] ) ->
  <png : File>
in Racket *{

  (require (only-in k-means/render
```

```
                  render-gen-sample)

          (only-in plot
                   plot))

  (define (proc f)
    (call-with-input-file f
      (lambda (in) (read in))))

  (define png "gen-sample.png")

  (plot (render-gen-sample (map proc cluster-lst))
        #:x-min      0
        #:x-max     10
        #:y-min      0
        #:y-max     10
        #:width    450
        #:height   450
        #:x-label  #f
        #:y-label  #f
        #:out-kind 'png
        #:out-file png)
}*

def plot-input-data( split-lst : [File] ) -> <png : File>
in Racket *{

  (require (only-in k-means/render
                   render-sample)

          (only-in plot
                   plot))

  (define (proc f)
    (call-with-input-file f
      (lambda (in) (read in))))

  (define png "input-data.png")

  (plot (render-sample (apply append (map proc split-lst)))
        #:x-min      0
        #:x-max     10
        #:y-min      0
        #:y-max     10
        #:width    450
        #:height   450
        #:x-label  #f
        #:y-label  #f
        #:out-kind 'png
```

```
       #:out-file png)
}*

def plot-history( split-lst : [File], history : [File] ) ->
  <png : File>
in Racket *{

  (require (only-in k-means/render
                    render-history)

           (only-in plot
                    plot))

  (define (proc f)
    (call-with-input-file f
      (lambda (in) (read in))))

  (define point-lst
    (apply append (map proc split-lst)))

  (define h
    (map proc history))

  (define png "history.png")

  (plot (render-history point-lst h)
        #:x-min      0
        #:x-max     10
        #:y-min      0
        #:y-max     10
        #:width    450
        #:height   450
        #:x-label  #f
        #:y-label  #f
        #:out-kind 'png
        #:out-file png)
}*

def plot-partition( split-lst : [File], cc-lst : File ) ->
  <png : File>
in Racket *{

  (require (only-in k-means/render
                    render-partition)

           (only-in plot
                    plot))

  (define (proc f)
```

```
    (call-with-input-file f
      (lambda (in) (read in))))

  (define point-lst
    (apply append (map proc split-lst)))

  (define ccl
    (proc cc-lst))

  (define png "partition.png")

  (plot (render-partition point-lst ccl)
        #:x-min      0
        #:x-max     10
        #:y-min      0
        #:y-max     10
        #:width    450
        #:height   450
        #:x-label  #f
        #:y-label  #f
        #:out-kind 'png
        #:out-file png)
}*

%%============================================================
%% Constants
%%============================================================

let k-gen  : Str = 4;
let n-gen  : Str = 10000;
let k-est  : Str = 4;
let n-part : Str = 8;

%%============================================================
%% Workflow
%%============================================================

let <cluster-lst = cluster-lst : [File]> =
  gen-random-sample(
    k = k-gen,
    n = n-gen );

let <split-lst = split-lst : [File]> =
  shuffle-split(
    cluster-lst = cluster-lst,
    n           = n-part );

let <cc-lst = cc-lst0 : File> =
  init-cc-lst( k = k-est );
```

```
let history : [File] =
  run-k-means( split-lst = split-lst,
               history   = [cc-lst0 : File] );

let cc-lst : File =
  first-file( lst     = history,
              default = error "empty list" : File );

let <png = random-sample-png : File> =
  plot-random-sample( cluster-lst = cluster-lst );

let <png = input-data-png : File> =
  plot-input-data( split-lst = split-lst );

let <png = history-png : File> =
  plot-history( split-lst = split-lst,
                history   = history );

let <png = partition-png : File> =
  plot-partition( split-lst = split-lst,
                  cc-lst    = cc-lst );

%%============================================================
%% Query
%%============================================================

<random-sample-png = random-sample-png,
 input-data-png    = input-data-png,
 history-png       = history-png,
 partition-png     = partition-png,
 cc-lst            = cc-lst>;
```

Erklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42 am 11. Juli 2018, habe ich zur Kenntnis genommen.

Declaration: I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt-Universität zu Berlin no. 42 on July 11 2018.

Berlin, January 2, 2021

Jörgen Brandt